ISO TC 184/SC4/WG11 N117 **Date: 2000-06-28**
Supersedes ISO TC 184/SC4/WG11 N110

ISO/CD 10303-14

Product data representation and exchange — Description methods: Part 14: The EXPRESS-X Language Reference Manual

ABSTRACT: This part of ISO 10303 specifies a language by which relationships between data defined by models in the EXPRESS language can be specified.

KEYWORDS: EXPRESS, Express-X, mapping language

COMMENTS TO READERS:
This part has been reviewed using the internal review checklist, the project leader check list and the convener check list, and has been determined to be ready for this ballot cycle

Project Leader: Martin Hardwick
Address: STEP Tools, Inc.,
216 River Street,
Troy, NY, 12180 USA
Telephone: +1-518-687-2848
Telefacsimile: +1-518-687-4420
Electronic Mail: Hardwick@steptools.com

Project Editor: Peter Denno
Address: NIST,
100 Bureau Drive,
Gaithersburg, MD, 20878 USA
Telephone: +1-301-975-3595
Telefacsimile: +1-301-975-4694
Electronic Mail: peter.denno@nist.gov

Contents

Page

Tables

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

International Standard ISO 10303-14 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data*.

A complete list of parts of ISO 10303 is available from the Internet:

http://www.nist.gov/sc4/editing/step/titles/

Annexes A, B and C form an integral part of this part of ISO 10303. Annexes D and E are for information only.

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

This International Standard is organized as a series of parts, each published separately. The parts of ISO 10303 fall into one of the following series: description methods, integrated resources, application interpreted constructs, application protocols, application modules, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303-1. This part of ISO 10303 is a member of the description methods series.

This part of ISO 10303 specifies the Express-X mapping language. It is expected that readers of this document understand the EXPRESS language, ISO 10303-11;1994 and ISO 10303-21;1994.

This specification provides industry with a means to document the relationship between information represented in EXPRESS.

INTERNATIONAL STANDARD (ISO) ISO/CD 10303-14

Industrial automation systems and integration — Product data representation and exchange — Part 14: Description methods: The EXPRESS-X language reference manual

1. Scope

This part of ISO 10303 specifies a language by which relationships between data defined by models in the EXPRESS language can be specified. The language is called EXPRESS-X.

EXPRESS-X is a structural data mapping language. It consists of language elements that allow an unambiguous specification of the relationship between models.

The following are within the scope of this part of ISO 10303:

– Mapping data defined by one EXPRESS model to data defined by another EXPRESS model.

– Mapping data defined by one version of an EXPRESS model to data defined by another version of an EXPRESS model, where the two schemas have different names.

– Specification of requirements for data translators for data sharing and data exchange applications.

– Specification of alternate views of data defined by an EXPRESS model.

– An alternate notation for application protocol mapping tables.

– Bi-directional mappings where mathematically possible.

– Specification of constraints evaluated against data produced by mapping.

The following are outside the scope of this part of ISO 10303:

– Mapping of data defined using means other than EXPRESS.

– Identification of the version of an EXPRESS schema.

– Graphical representation of constructs in the EXPRESS-X language.

2. Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO 10303-1:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles*.

ISO 10303-11:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.*

ISO 10303-21:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: Clear text encoding of exchange structure*.

3. Terms and Definitions

3.1 Terms defined in ISO 10303-1

For the purpose of this part of ISO 10303, the following terms defined in ISO 10303-1 apply:

– data;

– information;

– information model.

3.2 Terms defined in ISO 10303-11

For the purpose of this part of ISO 10303, the following terms defined in ISO 10303-11apply:

– complex entity data type;

– complex entity (data type) instance;

– constant;

– entity;

- entity data type;
- entity (data type) instance;
- instance;
- partial complex entity data type;
- partial complex entity value;
- population;
- simple entity (data type) instance;
- subtype/supertype graph;
- token;
- value.

3.3 Other definitions

For the purpose of this part of ISO 10303, the following definitions apply:

3.3.1 binding extent: a set of binding instances constructed from instances in the source data sets and view extents as identified by the FROM language element of a view/map declaration.

3.3.2 binding instance: an element of a binding extent.

3.3.3 map: the declaration of a relationship between data of one or more source entity types or view data types and data of one or more target entity types.

3.3.4 network mapping: a mapping to many target entity instances.

3.3.5 qualified binding extent: a subset of the binding extent consisting of only those binding instances satisfying the selection criteria of the view/map declaration.

3.3.6 selection criteria: EXPRESS logical expressions used to identify the qualified binding extent from a binding extent.

3.3.7 source data set: a collection of entity instances serving as an origin of mapping; each entity instance conforms to an entity data type defined in the associated schema, and the collection conforms to the constraints of the schema.

3.3.8 source extent: a view extent or entity population drawn on to create a binding extent.

3.3.9 target data set: a collection of entity instances produced by means of mapping.

3.3.10 view: an alternative organization of the information in an EXPRESS model.

3.3.11 view data type: the representation of a view.

3.3.12 view data type instance: a named unit of information that is a member of the view extent established by a view data type.

3.3.13 view extent: an aggregate of view data type instances that contains all instances that can be constructed from the qualified binding extent.

4. Conformance requirements

4.1 Formal specifications written in EXPRESS-X

4.1.1 Lexical language

A formal specification written in EXPRESS-X shall be consistent with a given level as specified below. A formal specification is consistent with a given level when all checks identified for that level as well as all lower levels are verified for the specification.

Levels of checking

Level 1: Reference checking. This level consists of checking the formal specification to ensure that it is syntactically and referentially valid. A formal specification is syntactically valid if it matches the syntax generated by expanding the primary syntax rule (`syntax`) given in Annex A. A formal specification is referentially valid if all references to EXPRESS-X items are consistent with the scope and visibility rules defined in clause 12.

Level 2: Type checking. This level consists of Level 1 checking and checking the formal specification to ensure that it is consistent with the following:

– expressions shall comply with the rules specified in clause 10 and in ISO 10303-11:1994 clause 12;

– assignments shall comply with the rules specified in ISO 10303-11:1994 clause 13.3.

Level 3: Value checking. This level consists of Level 2 checking and checking the formal specification to ensure that it is consistent with statements of the form, ‘A shall be greater than B’, as specified in clause 7 to 14 of ISO 10303-11:1994. This is limited to those places where both A and B can be evaluated from literals and/or constants.

Level 4: Complete checking. This level consists of checking the formal specification to ensure that it is consistent with all stated requirements as specified in this part of ISO 10303 and of ISO 10303-11:1994.

4.2 Implementations of EXPRESS-X

4.2.1 EXPRESS-X language parser

An implementation of an EXPRESS-X language parser shall be able to parse any formal specification written in EXPRESS-X consistent with the conformance class associated with that implementation. An EXPRESS-X language parser shall be said to conform to a particular level of checking (as defined in 4.1.1) if it can apply all checks required by that level (and any level below it) to a formal specification written in EXPRESS-X.

The implementor of an EXPRESS-X language parser shall state all constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented for the purpose of conformance testing.

4.2.2 EXPRESS-X mapping engine

An implementation of an EXPRESS-X mapping engine shall be able to evaluate and/or execute any formal specification written in EXPRESS-X, consistent with the conformance class associated with that implementation. The execution and/or evaluation of a mapping is relative to one or more source data sets; the specification of how these data sets are made available to the mapping engine is outside the scope of this part of ISO 10303.

The implementor of an EXPRESS-X mapping engine shall state any constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented for the purpose of conformance testing.

4.3 Conformance classes

An implementation shall be said to conform to conformance class 1 if it processes all the declarations that may appear in a SCHEMA_VIEW declaration.

An implementation shall be said to conform to conformance class 2 if it processes all the declarations that may appear in a SCHEMA_MAP declaration.

An implementation shall be said to conform to conformance class 3 if it processes all the declarations that may appear in this part of ISO 10303.

5. Language specification syntax

The notation used to present the syntax of the EXPRESS-X language is defined in this clause.

The full syntax for the EXPRESS-X language is given in Annex A. Portions of those syntax rules are reproduced in various clauses to illustrate the syntax of a particular statement. Those portions are not always complete. It will sometimes be necessary to consult Annex A for the missing rules. The syntax portions within this part of ISO 10303 are presented in a box. Each rule within the syntax box has a unique number toward the left margin for use in cross-references to other syntax rules.

The syntax of EXPRESS-X is defined in a derivative of Wirth Syntax Notation (WSN).

NOTE — See annex B for a reference describing Wirth Syntax Notation.

The notational conventions and WSN defined in itself are given below.

```
syntax= { production } .
production= identifier '=' expression '.' .
expression= term { '|' term } .
term= factor { factor } .
factor= identifier | literal | group | option | repetition .
identifier= character { character } .
literal= '''' character { character } '''' .
group= '(' expression ')' .
option= '[' expression ']' .
repetition= '{' expression '}' .
```

– The equal sign `'='` indicates a production. The element on the left is defined to be the combination of the elements on the right. Any spaces appearing between the elements of a production are meaningless unless they appear within a literal. A production is terminated by a period `'.'`.

– The use of an identifier within a factor denotes a nonterminal symbol that appears on the left side of another production. An identifier is composed of letters, digits, and the underscore character. The keywords of the language are represented by productions whose identifier is given in uppercase characters only.

– The word literal is used to denote a terminal symbol that cannot be expanded further. A literal is a sequence of characters enclosed in apostrophes. For an apostrophe to appear in a literal it must be written twice, i.e., `''''`.

- The semantics of the enclosing braces are defined below:
  - curly brackets `'{ }'` indicates zero or more repetitions;
  - square brackets `'[ ]'` indicates optional parameters;
  - parenthesis `'( )'` indicates that the group of productions enclosed by parenthesis shall be used as a single production;
  - vertical bar `'|'` indicates that exactly one of the terms in the expression shall be chosen.

The following notation is used to represent entire character sets and certain special characters which are difficult to display:

- **\a** represents any character from ISO/IEC 10646-1;
- **\n** represents a newline (system dependent) (see clause 7.1.5.2 of ISO 10303-11:1994).

6. Basic language elements

6.1 Overview

This clause specifies the basic elements from which an EXPRESS-X mapping specification is composed: the character set, remarks, symbols, reserved words, identifiers, and literals.

The basic language elements of EXPRESS-X are those of the EXPRESS language defined in Clause 7 of ISO 10303-11, with the exceptions noted below.

6.2 Reserved words

The reserved words of EXPRESS-X are the keywords and the names of built-in constants, functions, and procedures. Any reserved word in EXPRESS (ISO 10303-11:1994) shall also be a reserved word in EXPRESS-X. The reserved words shall not be used as identifiers. The additional reserved words of EXPRESS-X are described below.

In the case that a legal EXPRESS identifier is a reserved word in EXPRESS-X, schemas using that identifier can be mapped by renaming the conflicting identifier using the AS keyword in the REFERENCE language element.

In addition to the keywords of EXPRESS defined in ISO 10303-11:1994, the following are keywords of EXPRESS-X.

Table 1: Additional EXPRESS-X keywords

END_SCHEMA_MAP	EACH	SCHEMA_MAP	END_MAP
MAP	END_SCHEMA_VIEW	DEPENDENT_MAP	END_VIEW
SOURCE	IDENTIFIED_BY	TARGET	SCHEMA_VIEW
INDEXING	PARTITION	END_DEPENDENT_MAP	VIEW

7. Data types

7.1 Overview

The data types defined here as well as those defined in the EXPRESS language (clause 8 of ISO 10303-11:1994) are provided as part of the language.

Every view attribute has an associated data type.

7.2 View data type

View data types are established by view declarations (see clause 9.3). A view data type is assigned an identifier in the defining schema map or schema view. The view data type is referenced by this identifier.

Syntax:

```
229  view_reference = [ ( schema_map_ref | schema_view_ref ) '.'] view_ref .
```

Rules and restrictions:

a) view_ref shall be a reference to a view visible in the current scope.

b) view_ref shall not refer to a return view (clause 9.3.5).

EXAMPLE — following declaration defines a view data type named circle.

```
VIEW circle;
  FROM e : ellipse;
  WHERE (e.major_axis = e.minor_axis);
  SELECT
    radius : REAL := e.minor_axis;
    center : point := e.center;
END_VIEW;
```

8. Fundamental principles

8.1 Overview

The reader of this document is assumed to be familiar with the following concepts, in addition to the concepts described in clause 5 of ISO 10303-11:1994.

EXPRESS-X provides for the specification of:

– differing views of the data described by an information model described in EXPRESS;

– the transformation of data described by elements of source EXPRESS models into data described by elements of target EXPRESS models.

A `SCHEMA_MAP` provides declarations for the specification of the former and latter.

A `SCHEMA_VIEW` provides declarations for the specification of the former.

NOTE — A SCHEMA_VIEW may be transformed into an EXPRESS model as described in Annex B.

An EXPRESS-X schema may contain EXPRESS function and procedure specifications in order to support the definition of views, maps, or type maps.

8.2 Typographical conventions

In this specification a binding instance is denoted as an ordered set of entity / view instance name separated by commas “,” and enclosed in *angle brackets*, “<>”. Entity instance names are defined in ISO standard 10303 part(21) clause 7.3.4. View instance names are specified using the same syntax.

EXAMPLE — Given the view declaration:

```
VIEW example;
  FROM p: person; o : organization;
...
END_VIEW;
```

the following may be binding instances:

```
<#1,#31>
<#2,#32>.
```

These binding instances may correspond to the following data presented as entity instances as defined in ISO standard 10303 part (21):

```
#1=person('James','Smith');
#2=person('Fredrick','Jones');
#31=organization('Engineering');
#32=organization('Sales');
```

In this specification the data referenced by a binding extent may be presented in tabular form where the left-most column identifies the binding instance. The uppermost column headings, excluding the left-most column, identify express entity types or view data types. The lower headings identify the names of attributes corresponding to the entity identified in the uppermost column under which it falls, or when the heading cell contains ‘#’, the entity instance name.

EXAMPLE — This example illustrates the use of tables to depict a binding extent. The concept of a binding extent is defined in subsequent clauses and is not necessary to understand the example. The example uses the data defined in example 2 and the following EXPRESS schema:

```
SCHEMA example_3;
ENTITY person;
  first_name : STRING;
  last_name : STRING;
END_ENTITY;
ENTITY organization;
  department_name : STRING;
END_ENTITY;
END_SCHEMA;
```

Binding Instance	person			organization	
	#	first_name	last_name	#	department_name
<#1,#31>	#1	'James'	'Smith'	#31	'Engineering'
<#1,#32>	#1	'James'	'Smith'	#32	'Sales'
<#2,#31>	#2	'Fredrick'	'Jones'	#31	'Engineering'
<#2,#32>	#2	'Fredrick'	'Jones’	#32	'Sales’

8.3 Binding process

This specification defines a language and an execution model. The execution model is composed of two phases: a binding process and an instantiation process. The evaluation of views and maps share a common binding process but differ with respect to instantiation. A binding is an environment in which variables are given values during the instantiation process. Each binding instance provides a set of values to be assigned to the variables. The relationship between bindings and the source data is defined in subsequent clauses of this specification.

8.4 Implementation Environment

The EXPRESS-X language does not describe an implementation environment. In particular, EXPRESS-X does not specify:

– how references to names are resolved;

– how input and output data sets are specified;

– how mappings are executed for instances that do not conform to an EXPRESS schema.

The evaluation of a view (i.e. the application of the view to a source data set) produces a view extent. Evaluation of a map may produce entity instances in the target data set. EXPRESS-X does not specify what effect modification of source data may have on views and maps after their evaluation.

9. Declarations

9.1 Overview

This clause defines the various declarations available in EXPRESS-X. An EXPRESS-X declaration creates a new EXPRESS-X item and associates an identifier with it. The item may be referenced elsewhere by this identifier.

EXPRESS-X provides the following declarations:

– View;

– Map;

– Schema_view;

– Schema_map;

In addition, an EXPRESS-X specification may contain the following declarations defined in ISO 10303-11:1994:

– Constant;

– Function;

– Procedure;

– Rule.

9.2 Binding

9.2.1 Declaration of qualified binding extents

Syntax:

```
154  partition_header = [ PARTITION partition_id ;] from_clause [
     where_clause ] [ identified_by_clause ] .
```

A qualified binding extent is defined by identification and selection of binding instances.

The `FROM` language element defines the structure of instances in the binding extent. The FROM language element consists of one or more source_parameter. Each source parameter associates identifiers with an extent.

Syntax:

```
 89  from_clause = FROM source_parameter ';'{ source_parameter ';'} .
197  source_parameter = source_parameter_id ':' extent_reference .
```

Rules and restrictions:

a) `source_parameter_id`s shall be unique within the scope of the map or view declaration.

The binding extent is computed as the cartesian product of instances in the extents referenced in the FROM language element.

EXAMPLE 1 — A binding extent is constructed over the entity extents of entity types item and person.

```
SCHEMA example;  -- An EXPRESS schema
ENTITY item;
  item_number : INTEGER;
  approved_by : STRING;
END_ENTITY;
ENTITY person;
  name : STRING;
END_ENTITY;
END_SCHEMA;

VIEW items_and_persons
FROM i : item; p : person;
SELECT
 item_number : INTEGER := i.part_number;
 responsible : STRING := p.name;
END_VIEW;
```

Given a population (written as ISO 10303-21 entity instances):

```
#1=item(123,'Smith');
#2=item(234,'Smith');
#33=person('Jones');
#44=person('Smith');
```

the corresponding binding extent is: <#1,#33>,<#1,#44>,<#2,#33>,<#2,#44>.

The `WHERE` language element defines a selection criteria on binding instances. The `WHERE` language element, together with the source extents identified in the `FROM` language element define the qualified binding extent. A binding instance in the binding extent is a member of the qualified binding extent unless one or more domain rule expressions of the `WHERE` language element evaluates to FALSE for the application of that expression to the binding instance.

The syntax of the WHERE language element is as defined in ISO 10303-11;1994, clause 9.2.2.2.

EXAMPLE 2 — The qualified binding extent consists of those pairs of item and person of the binding extent for which person.name is 'Smith' or 'Jones' and item.approved_by is 'Smith' or 'Jones' and person.name = item.approved_by.

```
VIEW items_and_persons;
FROM i : item; p : person;
WHERE (p.name = 'Smith') OR (p.name = 'Jones');
      (i.approved_by = p.name);
SELECT
   name : STRING := p.name;
END_VIEW;
```

the corresponding qualified binding extent is: <#1,#44>,<#2,#44>.

9.2.2 Identification of view and target instances

The `IDENTIFIED_BY` declaration defines an equivalence relation between instances in a qualified binding extent.

Syntax:

```
107  identified_by_clause = IDENTIFIED_BY expression { ',' expression } ';'
     .
```

Rules and restrictions:

a) When used in a map declaration, an `expression` in an IDENTIFIED_BY language element shall not refer, through any level of indirection, to the targets of the map or any of their attributes.

Two qualified binding instances are in the same equivalence class if, for each expression of the IDENTIFIED_BY clause, evaluating the expression in the context of each of those instances produces result that are instance equal (ISO 10303-11;1994 clause 12.2.2). The instantiation process produces one view instance (views) or target network (maps) for each equivalence class.

EXAMPLE — This example illustrates the use of IDENTIFIED_BY.

```
VIEW department;
  FROM e : employee;
  IDENTIFIED_BY e.department_name;
  SELECT
    name : STRING := e.department_name;
END_VIEW;
ENTITY employee;
  name : STRING;
  department_name : STRING;
END_ENTITY;
...
END_VIEW;
```

```
#1=employee('Jones','Engineering');
#2=employee('Smith','Sales');
#3=employee('Doe','Engineering');
```

Given the view and population above, there are two equivalence classes: {#1,#3} and {#2}.

9.2.3 Equivalence classes and the instantiation process

View attributes (clause 9.3.2) and target entity attributes (clause 9.4.2) represent properties of the corresponding view (view) and target network entities (map). These attributes are provided values by evaluation of the corresponding expressions (`view_attr_assgnmt_expr` in views) (`map_attr_assgnmt_expr` in maps). The expressions are evaluated in the context of a binding instance in the qualified binding extent.

If an equivalence class defined by an IDENTIFIED_BY language element contains more than one qualified binding instance, then the value of the view_attr_assgnmt_expression is computed as follows:

– If for each such binding, the evaluation of the `view_attr_assgnmt_expr` (view) or `map_attr_assgnmt_expr` (map) of the attribute produces an equal value, that value is assigned to the attribute.

– If for two or more bindings, the evaluation of the `view_attr_assgnmt_expr` or `map_attr_assgnmt_expr` of the attribute produces unequal values, the indeterminate value is assigned to the attribute.

EXAMPLE — This example illustrates the assignment of values where an equivalence class contain more than one qualified binding instance. The map declaration is described in clause 9.4.

```
(* source schema *)            (* target schema *)
SCHEMA src;                   SCHEMA tar;
ENTITY employee;               ENTITY department;
  name : STRING;                 employee : STRING;
  manager : STRING;              manager : STRING;
  dept : STRING;                 name : STRING;
END_ENTITY;                    END_ENTITY;
END_SCHEMA;                     END_SCHEMA;
```

```
(* mapping schema *)
SCHEMA_MAP;
REFERENCE FROM src AS SOURCE;
REFERENCE FROM tar AS TARGET;
MAP department_map AS d : department
FROM e : src.employee
IDENTIFIED_BY e.dept;
SELECT
  d.name := e.dept;
  d.manager := e.manager;
  d.employee := e.name;
END_MAP;
END_SCHEMA_MAP;
```

```
#1=employee('Smith','Jones','Marketing');
#2=employee('Doe','Jones','Marketing');
```

Given the data above the target data set contains one entity instance, #1=department(?,’Jones','Marketing'). The attribute department.employee is indeterminate because the expression for this attribute evaluates to two different values ('Smith' and 'Doe').

9.3 View declaration

9.3.1 Overview

A view declaration creates a view data type and declares an identifier to refer to it.

EXAMPLE — The following view defines a view data type `arm_person_role_in_organization`.

```
VIEW arm_person_role_in_organization;
FROM pao : person_and_organization;
     ccdpaoa : cc_design_person_and_organization_assignment;
WHERE ccdpaoa.assigned_person_and_organization :=: pao;
SELECT
  person : person := pao.the_person;
  org : organization := pao.the_organizaion;
  role : label := ccdpaoa.role.name;
END_VIEW;
```

Syntax:

```
226 view_decl = VIEW view_id ':' base_type [ supertype_rule ] ';' (
    view_subtype_of_clause subtype_partition_header view_project_clause {
    subtype_partition_header view_project_clause } ) | (
    supertype_partition_header view_project_clause {
    supertype_partition_header view_project_clause } ) END_VIEW ';' .
154 partition_header = [ PARTITION partition_id ;] from_clause [
    where_clause ] [ identified_by_clause ] .
228 view_project_clause = ( SELECT view_attr_decl_stmt_list ) | ( RETURN
    expression ) .
```

Rules and restrictions:

a) If the view declaration specifies a `view_subtype_of_clause`, no `from_clause` shall be declared in any partition of the view declaration.

b) If the view declaration does not specify a `view_subtype_of_clause`, the `from_clause` is required in every partition of the view declaration.

c) Only a return view, clause 9.3.5, shall specify a base_type in `view_decl`.

9.3.2 View attributes

An attribute of a view data type represents a property of the view whose value is computed as the evaluation of its `view_attr_assgnmt_expr`, an expression.

The name of a view attribute (`view_attribute_id`) represents the role played by it associated value in the context of the view in which it appears.

Syntax:

```
228 view_project_clause = ( SELECT view_attr_decl_stmt_list ) | ( RETURN
    expression ) .
224 view_attr_decl_stmt_list = view_attribute_decl { view_attribute_decl }
    .
222 view_attribute_decl = view_attribute_id ':' [OPTIONAL] [
    source_schema_ref '.'] base_type ':=' expression ';' .
```

Rules and restrictions:

a) The `view_attr_assgnmt_expr` shall be assignment compatible with the data type of the view attribute.

b) Each view_attribute_id declared in the view declaration shall be unique within that declaration.

OPTIONAL indicates that the value of the attribute may be indeterminant. Use of OPTIONAL has no effect on the execution model.

9.3.3 View partitions

A view extent may be partitioned. The extent of a view that is partitioned is the concatenation of the extents defined by its partitions, each partition defining its own FROM language element and selection criteria. Partitions, if present, shall be named. A `partition_id` names a partition.

EXAMPLE — In ISO 10303-201, the application object `organization` may be mapped to either a `person`, an `organization`, or both a `person_and_organization` entity in the AIM. This is specified in EXPRESS-X as follows:

```
VIEW arm_organization;
PARTITION a_single_person;
  FROM p : person;
    ...
PARTITION a_single_organization;
  FROM o: organization;
    ...
PARTITION a_person_in_an_organization;
  FROM po: person_and_organization;
     ...
END_VIEW;
```

Syntax:

```
154 partition_header = [ PARTITION partition_id ;] from_clause [
    where_clause ] [ identified_by_clause ] .
```

Rules and restrictions:

a) All partitions of a `VIEW` declaration shall define the same attributes (including names and types)

b) The attributes of a `VIEW` declaration shall appear in the same order in each of its partitions.

9.3.4 Constant partitions

A partition that omits the FROM, WHERE, and IDENTIFIED_BY clauses is called a constant partition. Such a partition represents a single view instance in the result with no correspondence to the source data.

EXAMPLE — This example illustrates the use of constant partitions.

```
VIEW person;
PARTITION mary;
  SELECT
    name : STRING := 'Mary';
    age : INTEGER := 22;
PARTITION john;
   SELECT
    name : STRING := 'John';
    age : INTEGER := 23;
END_VIEW;
```

9.3.5 Return views

A `view_project_clause` defined as `RETURN expression` computes a value. The value shall not be of type AGGREGATE. The value computed shall be type compatible with `base_type`.

Syntax:

```
226 view_decl = VIEW view_id ':' base_type [ supertype_rule ] ';' (
    view_subtype_of_clause subtype_partition_header view_project_clause {
    subtype_partition_header view_project_clause } ) | (
    supertype_partition_header view_project_clause {
    supertype_partition_header view_project_clause } ) END_VIEW ';' .
228 view_project_clause = ( SELECT view_attr_decl_stmt_list ) | ( RETURN
    expression ) .
```

Rules and restrictions:

a) A return view shall not use the SELECT language element in any partition.

b) A return view shall not specify the view_subtype_of_clause language element

If an equivalence class defined by an IDENTIFIED_BY language element contains more than one qualified binding instance, then the value returned is computed as follows:

– If for each such binding, the RETURN expression produces an equal value, that value is returned.

– If for two or more bindings, the RETURN expression produces unequal values, the indeterminate value is returned.

A return view does not define a new type.

EXAMPLE 1 — This example defines instances of type car that have the value 'red' in their color attribute.

```
VIEW red_car;
  FROM rc:car;
  WHERE rc.color ='red';
  RETURN rc;
END_VIEW;
```

EXAMPLE 2 — This example defines an extent whose members are strings. The strings come from two sources.

```
VIEW owner_name : STRING;
  PARTITION one;
    FROM po:person;
    RETURN po.name;
  PARTITION two;
    FROM or: organization;
    RETURN or.name;
END_VIEW;
```

9.3.6 Specifying subtype views

EXPRESS-X allows for the specification of views as subtypes of other views, where a subtype view is a specialization of its supertype. This establishes an inheritance (i.e., subtype/supertype) relationship between the views in which the subtype inherits the properties (i.e., attributes and selection criteria) of its supertype. A view is a subtype view if it contains a `SUBTYPE` declaration. The extent of a subtype view is a subset of the extent of its supertype as defined by the selection criteria defined by the WHERE language element in the subtype.

A subtype view inherits attributes from its supertype view(s). Inheritance of attributes shall adhere to the rules and restrictions of attribute inheritance defined in ISO 10303-11;1994 clause 9.2.3.3.

A subtype view declaration may redefine attributes found in one of its supertypes. The redefinition of attributes shall adhere to the rules and restrictions of attribute redefinition defined in ISO 10303-11;1994 clause 9.2.3.4.

A view instance shall be created if the selection criteria of the most general supertype is satisfied. The view instance shall have the type corresponding to a subtype view if all of the selection criteria conditions in the subtype view in addition to all of its supertype views evaluate to TRUE or UNKNOWN.

Syntax:

```
230  view_subtype_of_clause = SUBTYPE OF '(' view_reference { ','
     view_reference } ')' .
```

Rules and restrictions:

a) A view declaration shall contain either a FROM language element or a subtype language element, but not both.

b) A subtype view shall not specify the IDENTIFIED_BY language element.

c) Exactly one supertype view of a subtype view shall define a FROM language element

d) The partitions of a subtype view shall be a subset of the partitions of its supertype view.

e) A subtype view shall not use the return language element.

EXAMPLE 1 — The following view illustrates subtyping. The view `male` defines an additional membership requirement (`gender = ‘M’`) for view instances of the subtype.

```
VIEW person;
FROM e:employee;
END_VIEW;

VIEW male SUBTYPE OF (person);
WHERE e.gender = 'M';
...
END_VIEW;
```

EXAMPLE 2 — This example illustrates the use of partitions and subtype views.

```
VIEW j;
PARTITION first;
FROM s:three, t:four
WHERE cond6;

PARTITION second;
FROM r:four, q:five
WHERE cond7;
END_VIEW;

VIEW k SUBTYPE OF (j);
PARTITION second;
WHERE cond9;
END_VIEW;
```

Any subtype view for which ‘k’ is a supertype can only include partition ‘second’.

9.3.7 SUPERTYPE constraints

A view declaration may define SUPERTYPE constraints (ISO standard 10303 part (11) clause 9.2.4). Whether or not a SUPERTYPE constraint is satisfied has no effect on the execution model or content of view extents.

EXAMPLE —

```
VIEW a ABSTRACT SUPERTYPE OF ONEOF(b ANDOR c, d);
 ...
END_VIEW;
```

An instance of 'a' is valid if it has at least two types ('a' and something else) because of the ABSTRACT keyword, and one of the other types is either 'd' or some combination of 'b' and 'c' because of the ONEOF keyword.

9.4 Map declaration

9.4.1 Overview

The MAP declaration supports the specification of correspondence between semantically equivalent elements of two or more EXPRESS models. The declaration supports the mapping from many source entities to many target entities.

Syntax:

```
135 map_decl = MAP map_id AS target_parameter ';'{ target_parameter ';'} (
    map_subtype_of_clause subtype_partition_header map_decl_body {
    subtype_partition_header [ map_decl_body ] } ) | (
    supertype_partition_header [ map_decl_body ] {
    supertype_partition_header map_decl_body } ) END_MAP ';' .
154 partition_header = [ PARTITION partition_id ;] from_clause [
    where_clause ] [ identified_by_clause ] .
136 map_decl_body = ( entity_instantiation_loop {
    entity_instantiation_loop } ) | map_project_clause | ( RETURN expres-
    sion ';' ).
211 target_parameter = [ target_parameter_id { ',' target_parameter_id }
    ':'] [ AGGREGATE [ bound_spec ] OF ] target_entity_reference .
```

Rules and restrictions:

a) If the map declaration contains more than one `map_partition`, each `map_partition` shall be named by a `partition_id` unique within the scope of the map declaration.

b) If a `target_parameter` does not specify a `target_parameter_id`, a `target_parameter_id` is implicitly defined and named as the `target_entitiy_ref`.

c) A map declaration containing the map_subtype_of_clause shall not contain more than one partitions.

`map_id` names a map declaration.

EXAMPLE — In the example below, a pump in the source data model is mapped to a product and product_related_product_category.

```
MAP network_for_pump AS pr : product;
                      prpc : product_related_product_category;
FROM p : pump;
 SELECT
  pr.id := p.id;
  pr.name := p.name;
  prpc.name := 'pump';
  prpc.products := [ pr ];
END_MAP;
```

The initial values of the attributes of the newly created instance(s) are indeterminate.

9.4.2 Evaluation of the MAP body

Syntax:

```
136  map_decl_body = ( entity_instantiation_loop {
     entity_instantiation_loop } ) | map_project_clause | ( RETURN expres-
     sion ';' ).
138  map_project_clause = SELECT map_attribute_declaration {
     map_attribute_declaration } .
132  map_attribute_declaration = [ target_parameter_ref [ index_qualifier ]
     [ group_qualifier ] '.'] attribute_ref [ index_qualifier ] ':='
     map_attr_assgnmt_expr ';' .
```

A map_body specifying `map_attribute_declaration`s shall assign values to the attributes of the target entity instances. The `map_attr_assgnmt_expr` shall produce a value that is assignment compatible with the target entity attribute (see ISO 10303-11;1994 clause 13.3).

A map_body specifying RETURN shall evaluate the expression which is specified after the RETURN keyword. Evaluation shall result in the instantiation of target entity instances that are type compatible with the entity types defined in the target_parameters.

9.4.3 Iteration under a single binding instance

9.4.3.1 Overview

Evaluation of a map may produce aggregates of target entity types. The initial value of the aggregate is indeterminant.

Syntax:

```
104 target_parameter = [ target_parameter_id { ',' target_parameter_id }
    ':' ] [ AGGREGATE [ bound_spec ] OF ] target_entity_reference ';' .
```

Rules and restrictions:

a) If `bound_spec` is specified it is treated as a constraint.

The `instantiation_loop_control` and `repeat_control` provides two mutually exclusive forms of iteration: iteration over the collection of instances in an EXPRESS aggregate; and interaction incrementing a numeric variable. The latter of these, provided by `repeat_control` is described in ISO 10303-11; 1994.;

Syntax:

```
 76 entity_instantiation_loop = FOR instantiation_loop_control ';'
    map_project_clause .
117 instantiation_loop_control = instantiation_foreach_control |
    repeat_control .
116 instantiation_foreach_control = EACH variable_id IN
    source_attribute_reference INDEXING variable_id { variable_id IN
    source_attribute_reference INDEXING variable_id } .
```

Rules and restrictions:

a) variable_id after the keyword EACH is of the same type as the elements of source_attribute_reference.

b) variable_id after the keyword INDEXING is of type NUMBER with values greater than one.

9.4.3.2 Control by numeric increment

The FOR repeat control allows for the iteration under a single binding instance by means of the EXPRESS `repeat_control`.

EXAMPLE — This example illustrates the use of the EXPRESS `repeat_control` in Express-X target instantiation. A collection of target child entity instances are created for each source parent entity. The number created is specified by the parent entity attribute number_of_children.

```
SCHEMA source;                    SCHEMA target;
ENTITY parent;                    ENTITY parent;
number_of_children : INTEGER;     END_ENTITY;
END_ENTITY;                       ENTITY child;
END_SCHEMA;                          parent : parent;
                                  END_ENTITY;
                                  END_SCHEMA;
```

```
SCHEMA_MAP example;
  REFERENCE FROM src AS SOURCE;
  REFERENCE FROM tar AS TARGET;
MAP tp AS tar.parent;
FROM sp : src.parent;
END_MAP;

MAP children_map AS c : AGGREGATE OF child;
FROM p : src.parent;
FOR i := 1 TO p.number_of_children
SELECT
  c[i].parent := p;
END_MAP;
END_SCHEMA_MAP;
```

9.4.3.3 Control by iteration over an aggregate

Under the `instantiation_foreach_control`, at each iteration step, the next element of the source attribute is bound to a variable and optionally the index position of that element is bound to an iterator variable. The scope of these variable bindings includes the `map_project_clause`.

EXAMPLE — In the following example, all item versions of one item are grouped together in the source data model. In the target model, each item version is an instance.

```
ENTITY item_version; --target data model
  item_id    : STRING;
  version_id : STRING;
END_ENTITY;

ENTITY item_with_versions; -- source data model
  id             : STRING;
  id_of_versions : LIST OF STRING;
END_ENTITY;
```

```
MAP iv : AGGREGATE OF item_version
FROM iwv : item_with_versions;
FOR EACH version_iterator OF iwv.id_of_versions INDEXING i
SELECT
  iv[i].item_id    := iwv.id;
  iv[i].version_id := version_iterator;
END_MAP;
```

For example, the following target instances are built from the source instance below.
Source instance set:

```
#1 = item_with_versions(1,(10,11,12));
```

Target instance set:

```
#1 = item_version(1,10);
#2 = item_version(1,11);
#3 = item_version(1,12);
```

An `instantiation_foreach_control` language element may specify many `source_attribute_references` using the optional AND syntax. Iteration continues while at least one source aggregate is not exhausted. The indeterminate value is assigned to the variable_id of exhausted aggregates.

9.4.4 Partitions within a MAP declaration

The instances of an entity type may each relate differently to source data. Multiple map partitions may be used to specify these differing relations.

If multiple target entities are listed in the header of the `MAP` declaration, different subsets of those entities may be created by each partition.

Syntax:

```
206  supertype_partition_header = [ PARTITION partition_id ';' ]
     from_clause [ where_clause ] [ identified_by_clause ].
```

Rules and restrictions:

a) The `partition_id` shall be unique with respect to the inheritance hierarchy of the corresponding target entity.

b) For every target entity declared in the map header, at least one partition shall be defined to create instances for it.

EXAMPLE — This example illustrates how various source entity types may be mapped into a single target entity type using a MAP declaration containing partitions.

```
(*  source schema *)
SCHEMA src;
ENTITY student;
     name : STRING;
END_ENTITY;
ENTITY employee;
  name : STRING;
END_ENTITY;
END_SCHEMA;
```

```
(* target schema *)
SCHEMA tar;
ENTITY person;
  name : STRING;
END_ENTITY;
END_SCHEMA;
```

```
(* mapping schema *)
SCHEMA_MAP example;
REFERENCE FROM src AS SOURCE;
REFERENCE FROM tar AS TARGET;
MAP student_employee_to_person AS p : tar.person;
PARTITION student;
FROM s : src.student;
SELECT
  p.name  := s.name;
PARTITION employee;
FROM e : src.employee;
SELECT
  p.name := e.name;
END_MAP;
```

9.4.5 Mapping to an entity type and its subtypes

EXPRESS-X allows for the specification of a map as a subtype of another map. Subtype map declarations may extend the collection of entity instances created by its supertype map, specialize those instances created and require additional selection criteria beyond those specified in the supertype map. The specification of a target attribute assignment declared in a supertype map is inherited by its subtype maps. Through these means, the pattern of inheritance present in the target schema can be duplicated in the mapping declarations.

Syntax:

```
141  map_subtype_of_clause = SUBTYPE OF '(' map_reference ')' ';'.
```

Whether a subtype map extends the collection of entity instances created by its supertype map or specializes those instance created depends on whether the subtype map references `target_parameter_id`s declared in the supertype map or whether it declare its own `target_parameter_id`s:

– If a map's selection criteria and that of all its supertype maps is satisfied, the map shall execute.

– A subtype map may reference a `target_parameter_id` that is declared in any of its supertype maps. The type created is the composition of types identified by the subtype map target parameter and all supertype maps declaring a target parameter with this target parameter id.

– A subtype map may introduce a `target_parameter_id` that is not defined in any of the supertype maps. In this case, a new target entity of the type defined by the target parameter is created.

A subtype map may reference for assignment a target attribute referenced for assignment in one of its supertypes (through possibly several levels of single inheritance). In this case, the target attribute is assigned the value corresponding to `map_attr_assgnnmt_expr` of the most specialized map for which the selection criteria and selection criteria of its supertypes is satisfied.

The type combination must be valid in the target schema.

A subtype map shall have exactly one direct supertype map.

NOTE — Multiple inheritance (i.e. a subtype map having more than one direct supertype map) is prohibited.

EXAMPLE — This example illustrates assignment to attributes declared in supertypes and subtypes through supertype and subtype maps. Source entities are of one type, `s_project`. Target entities are of type `t_project` and perhaps one of its subtypes, `in_house_project` and `external_project`. The target_parameter_id, `tp`, used in the supertype map (project_map) is used again in its subtype maps (in_house_map, ext_map) signifying that the corresponding target entity is specialized in the subtype maps.

```
SCHEMA source_schema;
ENTITY s_project;
  name : STRING;
  project_type : STRING;
  cost : INTEGER;
  price : INTEGER;
  vendor : STRING;
END_ENTITY;
END_SCHEMA;
```

```
SCHEMA target_schema;
ENTITY t_project;
SUPERTYPE OF (ONEOF (in_house_project, external_project));
  name : STRING;
  cost : INTEGER;
  management : STRING;
END_ENTITY;
ENTITY in_house_project;
SUBTYPE OF (t_project);
END_ENTITY;
ENTITY external_project;
SUBTYPE OF (t_project)
  price : INTEGER;
END_ENTITY;
END_SCHEMA;


SCHEMA_MAP example;
REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;
MAP project_map AS tp : target_schema.t_project;
FROM p : source_schema.s_project;
SELECT
     tp.name := p.name;
     tp.cost := p.cost;
END_MAP;

MAP in_house_map AS tp : target_schema.in_house_project;
SUBTYPE OF (project_map);
WHERE (p.project_type = ’in house’);
SELECT
  tp.management := IF (cost < 50000) THEN ’small accts’
                   ELSE ’large accts’ END_IF;
END_MAP;

MAP ext_map AS tp : target_schema.external_project;
SUBTYPE OF (project_map);
WHERE (p.project_type = ’external’);
SELECT
  tp.price := p.price;
  tp.management := p.vendor;
END_MAP;
```

A supertype map may define entity instantiation loops. A subtype map of such a supertype map shall inherit these instantiation loops. The correspondence between supertype map bodies and subtype map bodies where instantiation loops is made through use of identical index identifiers. The map body of the subtype map inheriting a loop shall reference the identical index identifier as defined in its supertype map.

EXAMPLE — This example illustrates the inheritance of an entity instantiation loop.

```
SCHEMA source_schema;
ENTITY part;
  name : STRING;
  no_of_versions : INTEGER;
  is_assembly : BOOLEAN;
END_ENTITY;
END_SCHEMA;

ENTITY target_schema;
ENTITY product;
  name : STRING;
END_ENTITY;

ENTITY product_version;
  version_id : INTEGER;
  of_product : product;
END_ENTITY;
ENTITY product_view;
  name : STRING;
  of_version : product_version;
END_ENTITY;

ENTITY assembly_view
SUBTYPE OF (product_view);
END_ENTITY;
END_SCHEMA;

SCHEMA_MAP example;
REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;
```

```
MAP super_map AS
  pvw   : AGGREGATE OF product_view;
  pver  : AGGREGATE OF product_version;
  pro    : product;
FROM prt : part;
FOR i := 1 TO no_of_versions;
SELECT
  pver[i].version_id := i;
  pver[i].of_product := p;
  pvw[i].of_version := pver[i];
  pvw[i].name := 'view of part ' + pro.name;
SELECT
  pro.name := 'part ' + part.name;
END_MAP;

MAP sub_map SUBTYPE OF (super_map)
  pvw    : AGGREGATE OF assembly_view;
  pver   : AGGREGATE OF product_version;
  pro    : product;
WHERE
  p.is_assembled = TRUE;
SELECT
  pvw[i].name := 'view of assembly ' + p.name;
SELECT
  p.name := 'assembly ' + part.name;
END_MAP;
END_SCHEMA_MAP;
```

9.4.6 Explicit declaration of complex entity data types

Complex entity data types (see ISO 10303-11:1994, clause 3.2.1) may be explicitly declared in the map header. A complex entity data type is referenced by an expression that lists the partial complex entity data types that are combined to form it, separated by ‘&’.

The partial complex entity data types may be listed in any order.

Any partial complex entity data types that are included in another partial complex entity data type via inheritance shall not be listed.

Syntax:

```
211 target_parameter = [ target_parameter_id { ',' target_parameter_id }
    ':'] [ AGGREGATE [ bound_spec ] OF ] target_entity_reference .
210 target_entity_reference = entity_reference { '&' entity_reference } .
 77 entity_reference = [ ( source_schema_ref | target_schema_ref |
    schema_ref ) '.'] entity_ref .
```

Rules and restrictions:

a) Each entity_ref shall be a reference to an entity which is visible in the current scope.

b) The referenced complex entity data type shall describe a valid domain within some schema (see ISO 10303-11:1994, annex B).

c) A given entity_ref shall occur at most once within a complex_entity_ref.

d) For each entity_reference declared in the complex_entity_spec, none of its supertype shall be declared.

9.4.7 Dependent map

A dependent map is a map that executes only when called as an explicit binding. A dependent map may have a simple type as a source parameter.

Syntax:

```
66  dependent_map_decl = DEPENDENT_MAP map_id AS target_parameter ';' {
    target_parameter ';' } [map_subtype_of_clause] dep_map_partition {
    dep_map_partition } END_DEPENDENT_MAP ';' .
70  dep_map_partition = [ PARTITION partition_id ':'] dep_map_decl_body
71  dep_map_decl_body = dep_binding_decl map_project_clause .
72  dep_binding_decl = dep_from_clause [where_clause]
    [identified_by_clause] .
73  dep_from_clause = FROM dep_source_parameter ';' { dep_source_parameter
    ';' } .
71  dep_source_parameter = source_parameter_id { ',' source_parameter_id}
    ':'(simple_types | type_reference) .
```

Rules and restrictions:

a) If more than one partition exists, a partition_id shall be provided for each partition.

b) Partition_ids shall be unique within the scope of the dependent_map.

EXAMPLE —

This example illustrates the use of a dependent map to instantiate target organization instances having unique id attributes. The call to the dependent map ensures organizations in the target population have unique id attributes.

```
MAP unique_orgs_map AS organization;
 PARTITION a_org;
 FROM a : named_organization;
   RETURN org_map(a.name);
PARTITION b_org;
 FROM b : id_organization;
   RETURN org_map(b.id);
 END_MAP;

DEPENDENT_MAP org_map AS org : organization;
 FROM id : STRING;
 SELECT
   org.id := id;
END_DEPENDENT_MAP;
```

9.5 Schema_view declaration

A schema_view declaration defines a common scope for a collection of related mapping declarations. A schema_view may contain the following kinds of declarations:

– constant declaration ();

– function declaration (clause 9.6);

– procedure declaration (clause 9.7);

– rule declarations (clause 9.11);

– view declaration (clause 9.3).

The order in which declarations appear within a schema_view declaration is not significant.

Declarations in one schema_view or EXPRESS schema may be made visible within the scope of another schema_view via an interface specification as described in clause 13.

Syntax:

```
187 schema_view_decl = SCHEMA_VIEW schema_view_id { reference_clause } [
    constant_decl ] schema_view_body_element_list END_SCHEMA_VIEW ';' .
166 reference_clause = REFERENCE FROM schema_ref_or_rename ['('
    resource_or_rename { ',' resource_or_rename } ')'] [ AS( SOURCE | TAR-
    GET) ] ';' .
186 schema_view_body_element_list = schema_view_body_element {
    schema_view_body_element } .
185 schema_view_body_element = function_decl | procedure_decl | view_decl
    | rule_decl .
```

Rules and restrictions:

a) The syntax `AS ( SOURCE | TARGET )` shall not be used in a schema_view_decl.

EXAMPLE — ap203_arm names a schema_view that may contain declarations defining a view over the schema config_control_design in terms of the domain expert's understanding of the information requirements.

```
SCHEMA_VIEW ap203_arm;
REFERENCE FROM config_control_design;
VIEW part_version ...
(* other declarations as appropriate *)
END_SCHEMA_VIEW;
```

9.6 Schema_map declaration

A `schema_map` declaration defines a common scope for a collection of related mapping declarations.

A `schema_map` may contain the following kinds of declarations:

- constant declaration (clause 9.5);
- function declaration (clause 9.6);
- procedure declaration (clause 9.7);
- view declaration (clause 9.3);
- map declaration (clause 9.4);
- rule declaration (clause 9.11).

The order in which declarations appear within a `schema_map` declaration is not significant.

Declarations in one `schema_map` may be made visible within the scope of another `schema_map` via an interface specification as described in clause 13.2.3

Syntax:

```
182 schema_map_decl = SCHEMA_MAP schema_map_id reference_clause {
    reference_clause } [ constant_decl ] schema_map_body_element_list
    END_SCHEMA_MAP ';' .
221 type_mapping_stmt = TYPE_MAP type_reference FROM type_reference ';'
    type_map_stmt_body type_map_stmt_body END_TYPE_MAP ';' .
180 schema_map_body_element = function_decl | procedure_decl | view_decl |
    map_decl | dependent_map_decl | create_map_decl | rule_decl .
```

Rules and restrictions:

a) The `schema_map` shall include explicitly or by import, at least one `MAP` declaration.

EXAMPLE 1 — `iges2step` names a `schema_map` that may contain declarations for translating geometry defined using an EXPRESS model base upon IGES into a model based on ISO 10303-203.

```
SCHEMA_MAP iges2step;
REFERENCE FROM step_schema AS TARGET;
REFERENCE FROM iges_express_schema AS SOURCE;
MAP iges_structure ...
(* other declarations as appropriate *)
END_SCHEMA_MAP;
```

A schema_map may reference EXPRESS schema, other schema_map schema and schema_view schema through use of the `reference_clause` language element. See clause 13.2.

Syntax:

```
182 schema_map_decl = SCHEMA_MAP schema_map_id reference_clause {
    reference_clause } [ constant_decl ] schema_map_body_element_list
    END_SCHEMA_MAP ';' .
166 reference_clause = REFERENCE FROM schema_ref_or_rename ['('
    resource_or_rename { ',' resource_or_rename } ')'] [ AS( SOURCE | TAR-
    GET) ] ';' .
184 schema_ref_or_rename = [ general_schema_alias_id ':']
    general_schema_ref .
```

Rules and restrictions:

a) A schema_map shall reference at least one EXPRESS schema designated as a mapping source using the AS SOURCE syntax.

b) A schema_map shall reference at least one EXPRESS schema designated as a mapping target using the AS TARGET syntax.

EXAMPLE 2 — This example illustrates the designation of source and target EXPRESS schema. EXPRESS schema t1 is referenced as the target of mapping. EXPRESS schema schema_source_one is referenced as the source of mapping; it may be referred to as s1 within the scope of this schema_map.

```
SCHEMA_MAP map_name;
  REFERENCE FROM t1 AS TARGET;
  REFERENCE FROM s1 : schema_source_one AS SOURCE;
END_SCHEMA_MAP;
```

9.7 Create declaration

The `CREATE` declaration defines the form of an entity that, subject to a logical expression, shall be created in the target data set. The `logical_expression` is evaluated against entity extents identified in the `target_entity_reference`. If the `logical_expression` evaluates to TRUE or if no `logical_expression` is specified, an entity shall be created in the target data set. If the logical_expression evaluates to the indeterminate value, the behaviour is undefined.

Syntax:

```
64 create_map_decl = CREATE instance_id ':' target_entity_reference ';' [
   WHERE logical_expression ';'] map_attribute_declaration {
   map_attribute_declaration } END_CREATE ';' .
```

Rules and restrictions:

a) `logical_expression` shall evaluate to either a `LOGICAL` value or indeterminate.

b) `target_entity_reference` shall refer to entity identifiers defined in a target schema.

c) Attribute references of the `map_attribute_declaration` shall refer to attributes of entities identified in the `target_entity_reference`.

EXAMPLE — In the following, an instance of application_context is created in the target data set provided that the entity extent of `item` (an entity type in a source schema) contains at least one instance.

```
CREATE APPCNT INSTANCE_OF application_context
WHERE SIZEOF(EXTENT(item)) > 0;
application := '';
END_CREATE;
```

9.8 Constant declaration

Constants may be defined for use within the WHERE language element of a view or map declaration, or within the body of a map declaration or algorithm.

Constant declarations are as defined in ISO 10303-11:1994 clause 9.4.

9.9 Function declaration

Functions may be defined for use within the WHERE language element of a view or map declaration, or within the body of a map declaration.

Function declarations are as defined in ISO 10303-11:1994 clause 9.5.1.

9.10 Procedure declaration

Procedures may be defined for use within the body of a map declaration.

Procedure declarations are as defined in ISO 10303-11:1994 clause 9.5.2.

9.11 Rule declaration

Rules may be defined for use within the SCHEMA_VIEW and SCHEMA_MAP language element.

Rule declarations are as defined in ISO 10303-11:1994 clause 9.6.

10. Expressions

10.1 Overview

Expressions are combinations of operators, operands, and function calls that are evaluated to produce a value.

The built-in function defined in Clause 15 and the operators defined in clause 12 of ISO 10303-11; 1994 apply to this specification. Arguments of type view shall be treated as arguments of type entity. The relationship between view definitions and entity definitions is defined in annex B.

Syntax:

```
 80 expression = simple_expression [ rel_op_extended simple_expression ] .
168 rel_op_extended = rel_op | IN | LIKE .
167 rel_op = '<' | ' > ' | '<=' | ' >= ' | ' <> ' | '=' | ':<>:' | ':=:' .
191 simple_expression = term { add_like_op term } .
214 term = factor { multiplication_like_op factor } .
 83 factor = simple_factor ['**' simple_factor ] .
192 simple_factor = aggregate_initializer | entity_constructor |
    enumeration_reference | interval | query_expression | ( [ unary_op ]
    ('(' expression ')' | primary ) ) | case_expr | for_expr .
157 primary = literal | ( qualifiable_factor { qualifier } ) .
162 qualifiable_factor = attribute_ref | constant_factor | function_call |
    population | general_or_map_call | view_call | view_attribute_ref .
```

Evaluation of an expression is governed by the precedence of the operators which form part of the expression. Expressions enclosed by parentheses are evaluated before being treated as a single operand.

Evaluation proceeds from left to right, with the highest precedence being evaluated first. Table 2 specifies the precedence rules for all of the operators of Express-X . Operators in the same row have the same precedence, and the rows are ordered by decreasing precedence. An operand between two operators of different precedence is bound to the operator with the higher precedence. An operand between two operators of the same precedence is bound to the one on the left.

Table 2: Operator precedence

Precedence	Description	Operators
1	Component Reference	[] . \ :: <- {}
2	Unary Operatators	+ - NOT
3	Exponentiation	**
4	Multiplication/Division	/ * DIV MOD AND ||
5	Addition/Subtraction	+ - OR XOR
6	Relational	= <> <= >= < > :=: :<>: IN LIKE

Entity constructors create instances that are local only to the function or procedure in which they are used. Instances produced by entity constructors shall not create target nor source populations.

10.2 View call

A view call is an expression that evaluates to a view instance or aggregate of view instances. The view call provides a means to access a view instance through arguments corresponding to its binding instance (when no IDENTIFIED_BY is defined) or IDENTIFIED_BY language element expressions (when IDENTIFIED_BY is defined). If no view instance corresponds, the call evaluates to indeterminate. A view call identifies a single partition of a view; if the view contains more than one partition, a partition_qualification shall be present. When no IDENTIFIED_BY language element is present in the partition, the number, type, and order of the actual parameters shall agree with that of the source parameters of the FROM language element in the partition. When an IDENTIFIED_BY language element is present, the number, type and order of the actual parameters shall agree with that of the expressions of the IDENTIFIED_BY language element.

A view call referencing a constant partition shall be passed an empty parameter list.

Syntax:

```
225 view_call = view_reference [ partition_qualification ] '('
    expression_or_wild { ',' expression_or_wild } ')' .
157 partition_qualification = '\' partition_ref .
 81 expression_or_wild = expression | '_' .
```

EXAMPLE — This example illustrates the use of a view call to define a relationship between two view data types. The IDENTIFIED_BY language element in the person_view specifies one expression, `a.creator`; view calls to person_view will therefore be supplied with one argument, a STRING which is also the creator attribute of an approval entity instance. The IDENTIFIED_BY clause in this view also serves to ensure the uniqueness of person_view instances (i.e. no two view instances will have the same name attribute).

```
SCHEMA_VIEW example;
VIEW approver
 PARTITION person_part;
  FROM a : approval; p : person;
  WHERE a.creator = p.name;
  IDENTIFIED_BY a.creator;
  SELECT
    approver_id : INTEGER := p.id;
 PARTITION org_part;
  FROM a : approval; o : organization;
  WHERE a.creator = o.name;
  IDENTIFIED_BY a.creator;
  SELECT
    approver_id : INTEGER := o.id;
END_VIEW;

VIEW design_order;
  FROM a : approval;
  SELECT
    id : STRING := a.id;
    approved_by : approver :=
                  approver\person_part(a.creator);
END_VIEW;
END_SCHEMA_VIEW;
```

```
SCHEMA src_schema;
ENTITY approval;
  id : STRING;
  creator : STRING;
END_ENTITY;
ENTITY person;
  name : STRING;
  id : INTEGER;
END_ENTITY;
END_SCHEMA;
(* Source data set in ISO 10303-21 form *)
#1=approval('a_1','Jones');
#2=approval('a_2','Smith');
#3=approval('a_3','Jones');
#4=person('Jones',123);
#5=person('Smith',234);

(* Resulting view instances in ISO 10303-21 form *)
#101=approver(123);
#102=approver(234);
#103=design_order('a_1',#101);
#104=design_order('a_2',#102);
#105=design_order('a_3',#101);
```

If one or more of the actual parameters is EXPRESS-X wildcard, '_', the result of the view call is an AGGREGATE containing those view instances of the view extent that correspond to the non-wildcard parameter values provided. If no view instances correspond, the view call evaluates to indeterminate.

EXAMPLE — In the following, the various versions associated with a part are collected by using a partial explicit binding. Returned by the explicit binding call version_and_its_product is the subset of the extent for which the second component of the binding is equal to the specified product instance.

```
VIEW part;
FROM (p : product)
SELECT
  versions : SET OF version_and_its_product
     := version_and_its_product(_, p);
END_VIEW;
```

10.3 Map call

A map call is an expression that evaluates to a target entity instance. A map call identifies a single partition of a map; if the map contains more than one partition, a partition_qualification shall be present. When no IDENTIFIED_BY language element is present in the partition or when the call is to a dependent map, the number, type, and order of the actual parameters shall agree with that of the source parameters of the FROM language element in the partition. When an IDENTIFIED_BY language element is present an the call is not to a dependent map, the number, type and order of the actual parameters shall agree with that of the expressions of the IDENTIFIED_BY language element.

Syntax:

```
100 general_or_map_call = general_ref ['@' map_call ] .
134 map_call = map_ref [ partition_qualification ] '(' expression_or_wild {
    ',' expression_or_wild } ')' .
157 partition_qualification = '\' partition_ref .
```

Rules and restrictions:

a) `target_parameter_ref` shall refer to a parameter reference declared in the MAP referenced as `map_ref`.

b) If the map declaration referenced by the map call declares more than one target parameter (i.e. it is a network map) the `general_ref @` syntax shall be used to identify the target entity to be returned by the map call.

EXAMPLE — This example illustrates the use of a map call to define a relationship between entities in the target schema.

```
(* source schema *)
SCHEMA src;
ENTITY approval;
  id : STRING;
  creator : STRING;
END_ENTITY;
END_SCHEMA;

SCHEMA tar;
ENTITY person;
  id : STRING;
END_ENTITY;
```

```
ENTITY design_order;
  id : STRING;
  approved_by : person;
END_ENTITY;
END_SCHEMA;

SCHEMA_MAP example;
REFERENCE FROM src AS SOURCE;
REFERENCE FROM tar AS TARGET;
MAP person_map AS p : target.person;
FROM a : approval
IDENTIFIED_BY a.creator
SELECT
  p.id := a.creator;
END_MAP;

MAP design_order_map AS d : target.design_order;
FROM a : approval
SELECT
  d.id := a.id;
  d.approved_by := p@person_map(a.creator); -- map call
END_MAP;
END_SCHEMA_MAP;
(* source instance set written as ISO 10303-21 instances *)
#1 = approval('a_1','miller');
#2 = approval('a_2','jones');
#3 = approval('a_3','miller');
(* Resulting target instances in ISO 10303-21 form *)
#101=person('Jones');
#102=person('Smith');
#103=design_order('a_1',#101);
#104=design_order('a_2',#102);
#105=design_order('a_3',#101);
```

10.4 Partial binding calls

A partial binding call is an view or map call in which one or more of the parameters is the EXPRESS-X wildcard '_'. The result of a partial binding call is an AGGREGATE that is the subset of the extent that matches the non-wildcard parameter values that are provided. If the subset is empty, the result of the partial binding call shall be indeterminate.

Partial binding calls to dependent maps are not permitted.

EXAMPLE — In the following, the various versions associated with a part are collected by using a partial explicit binding. Returned by the explicit binding call version_and_its_product is the subset of the extent for which the second component of the binding is equal to the specified product instance.

```
VIEW part;
FROM p : product;
SELECT
  versions : SET OF version_and_its_product
     := version_and_its_product(_, p);
END_VIEW;

VIEW version_and_its_product;
FROM (pdf : product_definition_formation, p : product)
WHERE p :=: pdf.of_product;
SELECT
  the_version : product_definition_formation := pdf;
END_VIEW;
```

10.5 FOR expression

The FOR expression collects the result of iteration of an expression over the elements of an EXPRESS aggregate. The iteration mechanism allows each element of the aggregate to be evaluated against a selection criteria. The collection is returned as an EXPRESS aggregate data type.

Syntax:

```
88  for_expr = FOR ( foreach_expr | forloop_expr ) .
84  foreach_expr = EACH variable_id IN expression { AND variable_id IN
    expression } [ where_clause ] RETURN expression .
85  forloop_expr = repeat_control RETURN expression .
```

Rules and restrictions:

a) Each `expression` of the `foreach_expr` shall evaluate to an EXPRESS aggregate, entity extent or view extent.

The iteration of the FOR expression is controlled either by the repeat_control (ISO 10303-11;1994 clause 13.9) or foreach_expr.

The EACH language element identifies an iterator variable. The IN language element identifies the EXPRESS aggregate or entity extent over which iteration shall occur. In each iteration of the loop an element of the aggregate is assigned to this iterator. The elements are selected in order proceeding from LOINDEX to HIINDEX (ISO 10303-11;1994 clauses 15.17 and 15.11).

The RETURN language element specifies an expression for each element during the iteration. All processed elements together build the result aggregate data type which is returned to the target attribute.

The optional `where_clause` specifies an expression that shall return a LOGICAL or indeterminate value. The expression following the RETURN language element is only evaluated when the where_clause returns TRUE.

EXAMPLE 1 — FOR expression.

```
(* Source schema *)
ENTITY product_definition;
  product_name : STRING;
  description : STRING;
END_ENTITY;

ENTITY product_definition_name;
  name    : STRING;
  of_product_definition : product_definition;
END_ENTITY;

(* Target schema *)
ENTITY component;
  names : SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
```

In this example, the target entity `component` maps to the source entity `product_definition` and all instances of `product_definition_name` which reference one instance of `product_definition` are grouped into the target attribute `component.names`. This is specified as follows.

```
Mapping definition:
MAP component AS mp : my_product;
FROM pd : product_definition
SELECT
  mp.description := pd.description;
  mp.product_name := pd.product_name;
  mp.names := FOR EACH pdn_instance
               IN pdn : product_definition_name
               WHERE pdn.of_product_definition :=: pd
             RETURN pdn_instance.name
END_MAP;
```

This example also shows that the scope of the FROM language element of the MAP declaration can be extended by the FROM language element of an FOR expression within this MAP declaration. That is, product_definition_name is not within the scope of the root entity of the FROM language element of the MAP declaration product_definition. In this case, the FOR expression specifies the outer join operation. That is, for each instance of product_definition a target instance of component is built independent of the existence of instances of product_definition_name which references this product_definition. If such instances of product_definition_name do not exist, the value of component.names is the empty set. Otherwise, those instances (e.g. the value product_definition_name.name) are assigned to the attribute component.names.

The RETURN language element can be nested in order to map attributes which are of type AGGREGATE OF AGGREGATE.

EXAMPLE 2 — Nested FOR expression. The example 31 is extended as follows.

```
Source schema:
ENTITY product_definition;
  (* as defined in Ex. 31 *)
END_ENTITY;

ENTITY product_definition_name;
  (* as defined in Ex. 31 *)
END_ENTITY;

ENTITY product_definition_value;
  of_pdn : product_definition_name;
  value : STRING;
END_ENTITY;

Target schema:
ENTITY component;
  values : SET [0:?] OF SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
```

In addition to example 1, all instances of `product_definition_value` which reference one instance of `product_definition_name` are grouped together and are assigned to the inner aggregate of `component.values`. This is specified as follows.

```
Mapping definition:
MAP component AS mp : my_product;
FROM pd : product_definition;
SELECT
  mp.description := pd.description;
  mp.product_name := pd.product_name;
  mp.names := FOR EACH pdn_instance
               IN pdn : product_definition_name;
               WHERE pdn.of_product_definition :=: pd;
           RETURN FOR EACH pdv_instance
                        IN pdv : product_definition_value;
                     WHERE pdv.of_pdn :=: pdn_instance;
                    RETURN pdv_instance.value;
END_MAP;
```

The FOR expression supports parallel iteration (i.e. iteration where two or more iterator variables are assigned to elements of sets). During each step of the iteration loop, all the iterator variables are assigned to the next element of the corresponding set. This is shown in the following example.

EXAMPLE 3 — Parallel iteration with the FOR expression.

```
Source schema:
ENTITY persons;
  firstname : SET [0:?] OF STRING;
  lastname : SET [0:?] OF STRING;
END_ENTITY;
```

```
Target schema:
ENTITY set_of_persons;
  name : SET [0:?] OF STRING;
END_ENTITY;
```

It is assumed that `persons.firstname[i]` corresponds to `persons.lastname[i]` and that those two values have to be concatenated and have to be assigned to `set_of_persons.name[i]`.

```
Mapping specification:
MAP set_of_persons_map AS p : set_of_persons;
FROM p : persons;
SELECT
  p.name := FOR EACH firstname_value IN p.firstname AND
                EACH lastname_value  IN p.lastname
            RETURN firstname_value + lastname_value;
END_MAP;
```

This example also shows that the FROM language element of the FOR expression is optional when it is a subset of the FROM language element of the MAP declaration. In this example, no predicates are needed to select specific elements of the extent which is given by the IN language element. Thus, the WHERE language element is omitted.

If the scope of the extent of the FOR loop (as specified by the foreach_in_clause_arg e.g., the repeat_control) is empty the FOR loop will be performed zero times.

10.6 IF expression

The if_expression is a map_attr_assgnmt_expr providing for the conditional evaluation of map_attr_assgnmt_exprs following the pattern of the EXPRESS IF statement (ISO 10303-11;1994 clause 13.7).

Syntax:

```
108 if_expr = IF logical_expression THEN map_attr_assgnmt_expr [ ELSE
    map_attr_assgnmt_expr ] END_IF .
```

10.7 CASE expression

The case_expr is a map_attr_assgnmt_expr providing for the conditional evaluation of map_attr_assgnmt_exprs following the pattern of the EXPRESS CASE statement (ISO 10303-11;1994, clause 13.4) .

Syntax:

```
 61  case_expr = CASE selector OF { case_expr_action } [ OTHERWISE ':'
     map_attr_assgnmt_expr ] END_CASE .
 58  case_expr_action = case_label { ',' case_label } ':'
     map_attr_assgnmt_expr .
```

EXAMPLE — CASE expression.

```
MAP approval_map AS a : my_approval
FROM a : approval
SELECT
  a.status := CASE a.status OF
              ’approved’     : 1;
              ’not approved’ : -1;
              ’indetermined’ : 0;
              OTHERWISE      : 2;
            END_CASE;
END_MAP;
```

10.8 Forward path operator

The forward path operator (::) provides an aggregate of entity instances referenced by or contained in attribute_ref. If the optional path_condition clause is specified, the result aggregate shall contain only entity instances of type entity_reference or of one of its subtypes. If additionally logical_expression is specified, the result shall only contain elements for which logical_expression evaluates to TRUE.

Syntax:

```
 87  forward_path_qualifier = '::' attribute_ref [ path_condition ] .
158  path_condition = '{' extent_reference ['|' logical_expression] '}' .
```

Rules and restrictions:

a) forward_path_qualifier shall not be used in conformance class 1 conforming schema_view;

For some entity extent `a`, an entity reference `product` and an attribute of instances in the extent `a`, `of_product`, the expression `result := a::of_product{product}` is equivalent to the following EXPRESS:

NOTE — The unnest function referenced below accepts one argument of arbitrary type (including a nested aggregate) and returns an aggregate whose elements are non-aggregate types. e.g. unnest([[a],[b,c],[[d]]]) returns [a,b,c,d]. See annex E for a definition of the unnest function used below.

```
LOCAL
  result : AGGREGATE OF GENERIC := [];
END_LOCAL;
tmp := unnest(a);
REPEAT i := 1 TO HIINDEX(tmp);
  result := result + QUERY(e <* unnest(tmp[i].of_product)|
                       'SCHEMA_NAME.PRODUCT' IN TYPEOF(e));
END_REPEAT;
result := unnest(result);
```

The expression `result := a::x` is equivalent to the EXPRESS:

```
result := [];
tmp := unnest(a);
REPEAT i := 1 TO HIINDEX(tmp);
  result := result + unnest(tmp[i].x);
END_REPEAT;
result := unnest(result)
```

10.9 Backward path operator

The backward_path_operator (<-) provides an aggregate of entity instances using the expression on the right side of the operator.

Syntax:

```
 46 backward_path_qualifier = '<-' [attribute_ref] path_condition .
158 path_condition = '{' extent_reference ['|' logical_expression] '}' .
```

Rules and restrictions:

a) backward_path_qualifier shall not be used in conformance class 1 conforming schema_view;

b) `attribute_ref` shall be defined in some partial entity type of each instance of the argument extent.

When identifier `a` represents an entity extent, the expression `result := a<-x{b}` is equivalent to the EXPRESS:

```
      result := [];
      tmp := unnest(a);
      REPEAT i := 1 TO HIINDEX(tmp);
         result := result +  QUERY(e <* USEDIN(tmp[i], '')  |
                               ('SCHEMA_NAME.B' IN TYPEOF(e))
                                AND (tmp[i] IN e.x));
      END_REPEAT;
```

The expression `a<-x` is equivalent to the EXPRESS:

```
result := [];
tmp := unnest(a);
REPEAT i := 1 TO HIINDEX(tmp);
   result := result + QUERY(e <* USEDIN(tmp[i],'') |
                        ('SCHEMA_NAME.B' IN TYPEOF(e)));
END_REPEAT;
```

EXAMPLE —

In this example path operators are used to compute the source aggregate of an instantiation loop. The source aggregate contains all instances of type document_file, referring to a representation_type instance with name 'digital' and are referenced as 'documentation_ids' of a 'product_definition_with_associated_documents' instance which refers to the source 'product_definition_formation' instance.

```
SCHEMA document_schema;

ENTITY folder;
   name : STRING;
END_ENTITY;

ENTITY file;
    name     : STRING;
    location : folder;
END_ENTITY;
END_SCHEMA;

SCHEMA_MAP example2;
TARGET document_schema;
SOURCE pdm_schema;
```

```
MAP document_map AS
  folder : folder;
  files:  LIST [0:?] OF file;
  FROM pdf : product_definition_formation;
    FOR EACH f IN
      pdf<-formation{product_definition_with_associated_documents}
            ::documentation_ids{document_file
            | representation_type.name = 'digital'}
     INDEXING i;
     SELECT
       files[i].name := f.name;
       files[i].location := folder;
       folder.name := pdf.name;
END_MAP;
```

11. Built-in functions

11.1 Extent - general function

`FUNCTION EXTENT ( R : STRING ) : SET OF GENERIC;`

The `EXTENT` function returns the population of instances of the type specified by the parameter.

Parameters:

a) R is a string that contains the name of a entity data type or view data type. Such names are qualified by the name of the schema which contains the definition of the type (‘SCHEMA.TYPE’).

Result: A set containing all instances of the entity data type or view data type specified in the parameter. It is an error to specify as the parameter a type which is neither a view data type nor an entity data type defined in a source schema.

12. Scope and visibility

An EXPRESS-X declaration creates an identifier that can be used to reference the declared item in other parts of the schema_view (or in other schema_views). Some EXPRESS-X constructs implicitly declare items, attaching identifiers to them. An item is said to be visible in those areas where an identifier for a declared item may be referenced. An item may only be referenced where its identifier is visible. For the rules of visibility, see clause 10.2 For further information on referring to items using their identifiers, see clause 12.

Certain EXPRESS-X items define a region (block) of text called the scope of the item. This scope limits the visibility of identifiers declared within it. Scope can be nested; that is, an EXPRESS-X item which establishes a scope may be included within the scope of another item. There are constraints on which items may appear within a particular EXPRESS-X item's scope.

For each of the items specified in table 2 below the following subclauses specify the limits of the scope defined, if any, and the visibility of the declared identifier both in general terms and with specific details.

Table 3: Scope and identifier defining items

Item	Scope	Identifier
view attribute		•
view	•	•
partition	•	•
schema_view	•	•

12.1 Scope rules

The general scope rules are as defined in ISO 10303-11:1994.

12.2 Visibility rules

The general visibility rules are as defined in ISO 10303-11:1994.

12.3 Explicit item rules

12.3.1 Overview

The following language elements provide more detail on how the general scoping and visibility rules apply to the various EXPRESS-X items.

12.3.2 Schema_view

Visibility: A schema_view identifier is visible to all other schema_views.

Scope: A schema_view declaration defines a new scope. This scope extends from the keyword SCHEMA_VIEW to the keyword END_SCHEMA_VIEW that terminates that schema_view declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a schema_view declaration:

– constant;

– function;

– procedure;

– rule;

– view.

12.3.3 View

Visibility: A view identifier is visible in the scope of the function, procedure, rule, or schema_view in which it is declared. A view identifier remains visible within inner scopes which redeclare that identifier.

Scope: A view declaration defines a new scope. This scope extends from the keyword VIEW to the keyword END_VIEW which terminates that entity declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a view declaration:

– view attribute;

– partition label.

12.3.4 View partition label

Visibility: A partition label is visible in the scope of the view in which it is declared.

12.3.5 View attribute identifier

Visibility: A view attribute identifier is visible in the scope of the view in which it is declared.

13. Interface specification

13.1 Overview

Interface specifications enable the reference of resources from external schemas, view_schemas and map schemas.

13.2 The reference language element

A REFERENCE specification enables the following items, declared in a foreign schema or schema_view, to be visible in the current schema_view:

– View;

– Constant;

– Entity;

– Type;

– Function;

– Procedure;

– Rule

Additionally, within a schema_map, the reference specification enables map declarations declared in a schema_map to be visible in the current schema_map.

The REFERENCE specification identifies the name of the foreign schema, schema_view or schema_map, and optionally the names of EXPRESS or EXPRESS-X items declared therein. If there are no names specified, all the items declared in the foreign schema or schema_view are visible within the current schema_view.

The schema_ref may be an EXPRESS-X reserved word provided that it is renamed by resource_or_rename.

Syntax:

```
166  reference_clause = REFERENCE FROM schema_ref_or_rename ['('
     resource_or_rename { ',' resource_or_rename } ')'] [ AS( SOURCE | TAR-
     GET) ] ';' .
184  schema_ref_or_rename = [ general_schema_alias_id ':']
     general_schema_ref .
173  resource_or_rename = resource_ref [ AS rename_id ] .
```

Rules and restrictions:

a) Within a schema_view, the `resource_or_rename` shall not reference a `map_ref`.

EXAMPLE — This example illustrates the reference of resources from other schema. The resource your_view_decl is referenced from the schema other_view_schema and is renamed my_view_decl for use within this schema_view.

```
SCHEMA_VIEW my_view_schema;
   REFERENCE FROM other_map_schema your_view_decl AS my_view_decl;
END_SCHEMA_MAP;
```

Annex A
(normative)
Information object registration

To provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(14) version(1) }

is assigned to this part of ISO 10303. The meaning of this value is defined is ISO/IEC 8824-1, and is described in ISO 10303-1.

Annex B
(normative)
EXPRESS-X language syntax

This annex defines the lexical elements of the language and the grammar rules that these elements shall obey.

NOTE — This syntax definition will result in ambiguous parsers if used directly. It has been written so as to convey information regarding the use of identifiers. The interpreted identifiers define tokens that are references to declared identifiers, and therefore should not resolve to simple_id. This requires a parser developer to enable identifier reference resolution and return the required reference token to a grammar rule checker.

All of the grammar rules of EXPRESS specified in annex A of ISO 10303-11:1994 are also grammar rules of EXPRESS-X. In addition, the grammar rules specified in the remainder of this annex are grammar rules of EXPRESS-X.

B.1 Tokens

The following rules specify the tokens used in EXPRESS-X. Except where explicitly stated in the syntax rules, no white space or remarks shall appear within the text matched by a single syntax rule in the following clauses.

B.1.1 Keywords

This subclause gives the rules used to represent the keywords of EXPRESS-X.

NOTE — This subclause follows the typographical convention that each keyword is represented by a syntax rule whose left hand side is that keyword in uppercase.

NOTE — All the keywords of EXPRESS are also keywords of EXPRESS-X

```
 1 CREATE = 'create'.
 2 DEPENDENT_MAP = 'dependent_map'.
 3 EACH = 'each'.
 4 ELSIF = 'elsif' .
 5 END_CHOICE = 'end_choice' .
 6 END_CREATE = 'end_create'.
 7 END_DEPENDENT_MAP = 'end_dependent_map'.
 8 END_FOR = 'end_for'.
 9 END_MAP = 'end_map'.
10 END_SCHEMA_MAP = 'end_schema_map'.
11 END_SCHEMA_VIEW = 'end_schema_view'.
12 END_VIEW = 'end_view'.
13 EXTENT = 'extent' .
```

```
14 IDENTIFIED_BY = 'identified_by'.
15 MAP = 'map'.
16 PARTITION = 'partition'.
17 SCHEMA_MAP = 'schema_map'.
18 SCHEMA_VIEW = ’schema_view’.
19 SOURCE = 'source'.
20 TARGET = 'target'.
21 VIEW = 'view'.
```

B.1.2 Character classes

```
22 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
23 letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
          | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
          | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .
24 simple_id = letter { letter  | digit  | '_' } .
```

B.1.3 Interpreted identifiers

NOTE — All interpreted identifiers of EXPRESS are also interpreted in EXPRESS-X

```
25 instance_ref = instance_id .
26 network_ref = network_id .
27 partition_ref = partition_id .
28 schema_map_ref = schema_map_id .
29 schema_view_ref = schema_view_id .
30 source_schema_ref = schema_ref .
31 target_schema_ref = schema_ref .
32 view_attribute_ref = view_attribute_id .
33 view_ref = view_id .
```

B.2 Grammar rules

```
34 actual_parameter_list = '(' parameter { ',' parameter } ')' .
35 add_like_op = '+' | '-' | OR | XOR .
36 aggregate_initializer = '[' [ element { ',' element } ] ']' .
37 aggregate_source =  simple_expression .
38 aggregate_type =  AGGREGATE [':' type_label ] OF parameter_type .
39 aggregation_types =  array_type | bag_type | list_type | set_type .
40 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
41 array_type =  ARRAY bound_spec OF [ OPTIONAL ] [ UNIQUE ] base_type .
42 assignment_stmt =  general_ref { qualifier } ':=' expression ';' .
43 backward_path_qualifier = '<-' [attribute_ref] path_condition .
44 bag_type =  BAG [ bound_spec ] OF base_type .
45 base_type =  aggregation_types | simple_types | named_types .
```

46 binary_type = BINARY [width_spec] .

47 boolean_type = BOOLEAN .

48 bound_1 = numeric_expression .

49 bound_2 = numeric_expression .

50 bound_spec = '[' bound_1 ':' bound_2 ']' .

51 built_in_constant = CONST_E | PI | SELF | '?' .

52 built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS | extent | EXP | FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND | LOINDEX | LOG | LOG2 | LOG10 | NVL | ODD | ROLESOF | SIN | SIZEOF | SQRT | TAN | TYPEOF | USEDIN | VALUE | VALUE_IN | VALUE_UNIQUE .

53 built_in_procedure = INSERT | REMOVE .

54 case_action = case_label { ',' case_label } ':' stmt .

55 case_expr = CASE selector OF { case_expr_action } [OTHERWISE ':' map_attr_assgnmt_expr] END_CASE .

56 case_expr_action = case_label { ',' case_label } ':' map_attr_assgnmt_expr ';' .

57 case_label = expression .

58 case_stmt = CASE selector OF { case_action } [OTHERWISE ':' stmt] END_CASE ';' .

59 compound_stmt = BEGIN stmt { stmt } END ';' .

60 constant_body = constant_id ':' base_type ':=' expression ';' .

61 constant_decl = CONSTANT constant_body { constant_body } END_CONSTANT ';' .

62 constant_factor = built_in_constant | constant_ref .

63 constant_id = simple_id .

64 create_map_decl = CREATE instance_id ':' target_entity_reference ';' [WHERE logical_expression ';'] map_attribute_declaration { map_attribute_declaration } END_CREATE ';' .

65 declaration = function_decl | procedure_decl .

66 dependent_map_decl = DEPENDENT_MAP map_id AS target_parameter ';' { target_parameter ';' } [map_subtype_of_clause] dep_map_partition { dep_map_partition } END_DEPENDENT_MAP ';' .

67 dep_binding_decl = dep_from_clause [where_clause] [identified_by_clause] .

68 dep_from_clause = FROM dep_source_parameter ';' { dep_source_parameter ';' } .

69 dep_map_decl_body = dep_binding_decl map_project_clause .

70 dep_map_partition = [PARTITION partition_id ':'] dep_map_decl_body

71 dep_source_parameter = source_parameter_id { ',' source_parameter_id} ':'(simple_types | type_reference) .

72 domain_rule = label ':' logical_expression .

73 element = expression [':' repetition] .

74 entity_constructor = entity_ref '(' [expression { ',' expression }] ')' .

75 entity_id = simple_id .

76 entity_instantiation_loop = FOR instantiation_loop_control ';' map_project_clause .

```
77  entity_reference = [ ( source_schema_ref | target_schema_ref |
    schema_ref ) '.'] entity_ref .
78  enumeration_reference = [ type_ref '.'] enumeration_ref .
79  escape_stmt = ESCAPE ';' .
80  expression = simple_expression [ rel_op_extended simple_expression ] .
81  expression_or_wild = expression | '_' .
82  extent_reference = source_entity_reference | view_reference .
83  factor = simple_factor ['**' simple_factor ] .
84  foreach_expr = EACH variable_id IN expression { AND variable_id IN
    expression } [ where_clause ] RETURN expression .
85  forloop_expr = repeat_control RETURN expression .
86  formal_parameter = parameter_id { ',' parameter_id } ':'
    parameter_type .
87  forward_path_qualifier = '::' attribute_ref [ path_condition ] .
88  for_expr = FOR ( foreach_expr | forloop_expr ) .
89  from_clause = FROM source_parameter ';'{ source_parameter ';'} .
90  function_call = ( built_in_function | function_ref ) [
    actual_parameter_list ] .
91  function_decl = function_head [ algorithm_head ] stmt { stmt }
    END_FUNCTION ';' .
92  function_head = FUNCTION function_id ['(' formal_parameter { ';'
    formal_parameter } ')'] ':' parameter_type ';' .
93  function_id = simple_id .
94  generalized_types = aggregate_type | general_aggregation_types |
    generic_type .
95  general_aggregation_types = general_array_type | general_bag_type |
    general_list_type | general_set_type .
96  general_array_type = ARRAY [ bound_spec ] OF [ OPTIONAL ] [ UNIQUE ]
    parameter_type .
97  general_attribute_qualifier = '.' ( attribute_ref | view_attribute_ref
    ) .
98  general_bag_type = BAG [ bound_spec ] OF parameter_type .
99  general_list_type = LIST [ bound_spec ] OF [ UNIQUE ] parameter_type .
100 general_or_map_call = general_ref ['@' map_call ] .
101 general_ref = parameter_ref | variable_ref .
102 general_schema_alias_id = schema_id | schema_map_id | schema_view_id .
103 general_schema_ref = schema_ref | schema_map_ref | schema_viw_ref .
104 general_set_type = SET [ bound_spec ] OF parameter_type .
105 generic_type = GENERIC [':' type_label ] .
106 group_qualifier = '\' entity_ref .
107 identified_by_clause = IDENTIFIED_BY expression { ',' expression } ';'
    .
108 if_expr = IF logical_expression THEN map_attr_assgnmt_expr [ ELSE
    map_attr_assgnmt_expr ] END_IF .
109 if_stmt = IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt
    } ] END_IF ';' .
```

```
110 increment = numeric_expression .
111 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment
    ] .
112 index = numeric_expression .
113 index_1 = index .
114 index_2 = index .
115 index_qualifier = '[' index_1 [':' index_2 ] ']' .
116 instantiation_foreach_control = EACH variable_id IN
    source_attribute_reference INDEXING variable_id { variable_id IN
    source_attribute_reference INDEXING variable_id } .
117 instantiation_loop_control = instantiation_foreach_control |
    repeat_control .
118 integer_type = INTEGER .
119 interval = '{' interval_low interval_op interval_item interval_op
    interval_high '}' .
120 interval_high = simple_expression .
121 interval_item = simple_expression .
122 interval_low = simple_expression .
123 interval_op = '<' | '<=' .
124 label = simple_id .
125 list_type = LIST [ bound_spec ] OF [ UNIQUE ] base_type .
126 literal = binary_literal | integer_literal | logical_literal |
    real_literal | string_literal .
127 local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .
128 local_variable = variable_id { ',' variable_id } ':' parameter_type
    [':=' expression ] ';' .
129 logical_expression = expression .
130 logical_literal = FALSE | TRUE | UNKNOWN .
131 logical_type = LOGICAL .
132 map_attribute_declaration = [ target_parameter_ref [ index_qualifier ]
    [ group_qualifier ] '.'] attribute_ref [ index_qualifier ] ':='
    map_attr_assgnmt_expr ';' .
133 map_attr_assgnmt_expr = expression | if_expr | case_expr | for_expr .
134 map_call = map_ref [ partition_qualification ] '(' expression_or_wild
    { ',' expression_or_wild } ')' .
135 map_decl = MAP map_id AS target_parameter ';'{ target_parameter ';'} (
    map_subtype_of_clause subtype_partition_header map_decl_body {
    subtype_partition_header [ map_decl_body ] } ) | (
    supertype_partition_header [ map_decl_body ] {
    supertype_partition_header map_decl_body } ) END_MAP ';' .
136 map_decl_body = ( entity_instantiation_loop {
    entity_instantiation_loop } ) | map_project_clause | ( RETURN expres-
    sion ';' ).
137 map_id = simple_id .
138 map_project_clause = SELECT map_attribute_declaration {
    map_attribute_declaration } .
139 map_ref = map_id .
```

```
140 map_reference = [ schema_map_ref '.'] map_ref .
141 map_subtype_of_clause = SUBTYPE OF '(' map_reference ')' ';'.
142 multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
143 named_types = entity_ref | type_ref | view_ref .
144 null_stmt = ';' .
145 number_type = NUMBER .
146 numeric_expression = simple_expression .
147 one_of = ONEOF '(' supertype_expression { ',' supertype_expression }
    ')' .
148 parameter = expression .
149 parameter_id = simple_id .
150 parameter_type = generalized_types | named_types | simple_types .
151 partition_id = simple_id .
152 partition_qualification = '\' partition_ref .
153 path_condition = '{' extent_reference ['|' logical_expression] '}' .
154 path_qualifier = forward_path_qualifier | backward_path_qualifier .
155 population = entity_ref .
156 precision_spec = numeric_expression .
157 primary = literal | ( qualifiable_factor { qualifier } ) .
158 procedure_call_stmt = ( built_in_procedure | procedure_ref ) [
    actual_parameter_list ] ';' .
159 procedure_decl = procedure_head [ algorithm_head ] { stmt }
    END_PROCEDURE ';' .
160 procedure_head = PROCEDURE procedure_id ['(' [ VAR ] formal_parameter
    { ';' [ VAR ] formal_parameter } ')'] ';' .
161 procedure_id = simple_id .
162 qualifiable_factor = attribute_ref | constant_factor | function_call |
    population | general_or_map_call | view_call | view_attribute_ref .
163 qualifier = general_attribute_qualifier | group_qualifier |
    index_qualifier | view_attribute_qualifier | path_qualifier .
164 query_expression = QUERY '(' variable_id ' <* ' aggregate_source '|'
    logical_expression ')' .
165 real_type = REAL ['(' precision_spec ')'] .
166 reference_clause = REFERENCE FROM schema_ref_or_rename ['('
    resource_or_rename { ',' resource_or_rename } ')'] [ AS( SOURCE | TAR-
    GET) ] ';' .
167 rel_op = '<' | ' > ' | '<=' | ' >= ' | ' <> ' | '=' | ':<>:' | ':=:' .
168 rel_op_extended = rel_op | IN | LIKE .
169 rename_id = constant_id | entity_id | function_id | procedure_id |
    type_id .
170 repeat_control = [ increment_control ] [ while_control ] [
    until_control ] .
171 repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .
172 repetition = numeric_expression .
173 resource_or_rename = resource_ref [ AS rename_id ] .
```

```
174 resource_ref =  constant_ref | entity_ref | function_ref |
    procedure_ref | type_ref | view_ref | map_ref .
175 return_stmt =  RETURN ['(' expression ')'] ';' .
176 rule_decl =  rule_head [ algorithm_head ] { stmt } where_clause
    END_RULE ';' .
177 rule_head =  RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')'';'
    .
178 rule_id =  simple_id .
179 schema_id =  simple_id .
180 schema_map_body_element =  function_decl | procedure_decl | view_decl |
    map_decl | dependent_map_decl | create_map_decl | rule_decl .
181 schema_map_body_element_list =  schema_map_body_element {
    schema_map_body_element } .
182 schema_map_decl =  SCHEMA_MAP schema_map_id reference_clause {
    reference_clause } [ constant_decl ] schema_map_body_element_list
    END_SCHEMA_MAP ';' .
183 schema_map_id =  simple_id .
184 schema_ref_or_rename =  [ general_schema_alias_id ':']
    general_schema_ref .
185 schema_view_body_element =  function_decl | procedure_decl | view_decl
    | rule_decl .
186 schema_view_body_element_list =  schema_view_body_element {
    schema_view_body_element } .
187 schema_view_decl =  SCHEMA_VIEW schema_view_id { reference_clause } [
    constant_decl ] schema_view_body_element_list END_SCHEMA_VIEW ';' .
188 schema_view_id =  simple_id .
189 selector =  expression .
190 set_type =  SET [ bound_spec ] OF base_type .
191 simple_expression =  term { add_like_op term } .
192 simple_factor =  aggregate_initializer | entity_constructor |
    enumeration_reference | interval | query_expression | ( [ unary_op ]
    ('(' expression ')' | primary ) ) | case_expr | for_expr .
193 simple_types =  binary_type | boolean_type | integer_type |
    logical_type | number_type | real_type | string_type .
194 skip_stmt =  SKIP ';' .
195 source_attribute_reference =  parameter_ref '.' ( attribute_ref |
    view_attribute_ref ) .
196 source_entity_reference =  entity_reference .
197 source_parameter =  source_parameter_id ':' extent_reference .
198 source_parameter_id =  parameter_id .
199 stmt =  assignment_stmt | case_stmt | compound_stmt | escape_stmt |
    if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
    skip_stmt .
200 string_literal =  simple_string_literal | encoded_string_literal .
201 string_type =  STRING [ width_spec ] .
202 subtype_constraint =  OF '(' supertype_expression ')' .
203 subtype_partition_header = [ PARTITION partition_id ';'] where_clause .
```

```
204 supertype_expression = supertype_factor { ANDOR supertype_factor } .
205 supertype_factor = supertype_term { AND supertype_term } .
206 supertype_partition_header = [ PARTITION partition_id ';' ] from_clause
    [ where_clause ] [ identified_by_clause ].
207 supertype_rule = [ ABSTRACT ] SUPERTYPE [ subtype_constraint ] .
208 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
209 syntax = schema_map_decl | schema_view_decl .
210 target_entity_reference = entity_reference { '&' entity_reference } .
211 target_parameter = [ target_parameter_id { ',' target_parameter_id }
    ':'] [ AGGREGATE [ bound_spec ] OF ] target_entity_reference .
212 target_parameter_id = parameter_id .
213 target_parameter_ref = target_parameter_id .
214 term = factor { multiplication_like_op factor } .
215 type_id = simple_id .
216 type_label = type_label_id | type_label_ref .
217 type_label_id = simple_id .
218 type_reference = [ schema_ref '.'] type_ref .
219 unary_op = '+' | '-' | NOT .
220 until_control = UNTIL logical_expression .
221 variable_id = simple_id .
222 view_attribute_decl = view_attribute_id ':' [OPTIONAL] [
    source_schema_ref '.'] base_type ':=' expression ';' .
223 view_attribute_id = simple_id .
224 view_attr_decl_stmt_list = view_attribute_decl { view_attribute_decl }
    .
225 view_call = view_reference [ partition_qualification ] '('
    expression_or_wild { ',' expression_or_wild } ')' .
226 view_decl = VIEW view_id ':' base_type [ supertype_rule ] ';' (
    view_subtype_of_clause subtype_partition_header view_project_clause {
    subtype_partition_header view_project_clause } ) | (
    supertype_partition_header view_project_clause {
    supertype_partition_header view_project_clause } ) END_VIEW ';' .
227 view_id = simple_id .
228 view_project_clause = ( SELECT view_attr_decl_stmt_list ) | ( RETURN
    expression ) .
229 view_reference = [ ( schema_map_ref | schema_view_ref ) '.'] view_ref
    .
230 view_subtype_of_clause = SUBTYPE OF '(' view_reference { ','
    view_reference } ')' .
231 where_clause = WHERE domain_rule ';'{ domain_rule ';'} .
232 while_control = WHILE logical_expression .
233 width = numeric_expression .
234 width_spec = '(' width ')' [ FIXED ] .
```

B.3 Cross reference listing

```
 34  actual_parameter_list             | 90 158
 35  add_like_op                       | 191
 36  aggregate_initializer             | 192
 37  aggregate_source                  | 164
 38  aggregate_type                    | 94
 39  aggregation_types                 | 45
 40  algorithm_head                    | 91 159 176
 41  array_type                        | 39
 42  assignment_stmt                   | 199
 43  backward_path_qualifier           | 154
 44  bag_type                          | 39
 45  base_type                         | 41 44 60 125 190 222 226
 46  binary_type                       | 193
 47  boolean_type                      | 193
 48  bound_1                           | 50 111
 49  bound_2                           | 50 111
 50  bound_spec                        | 41 44 96 98 99 104 125 190 211
 51  built_in_constant                 | 62
 52  built_in_function                 | 90
 53  built_in_procedure                | 158
 54  case_action                       | 58
 55  case_expr                         | 133 192
 56  case_expr_action                  | 55
 57  case_label                        | 54 56
 58  case_stmt                         | 199
 59  compound_stmt                     | 199
 60  constant_body                     | 61
 61  constant_decl                     | 40 182 187
 62  constant_factor                   | 162
 63  constant_id                       | 60 169
 64  create_map_decl                   | 180
 65  declaration                       | 40
 66  dependent_map_decl                | 180
 67  dep_binding_decl                  | 69
 68  dep_from_clause                   | 67
 69  dep_map_decl_body                 | 70
 70  dep_map_partition                 | 66
 71  dep_source_parameter              | 68
 72  domain_rule                       | 231
 73  element                           | 36
 74  entity_constructor                | 192
 75  entity_id                         | 169
 76  entity_instantiation_loop         | 136
 77  entity_reference                  | 196 210
 78  enumeration_reference             | 192
 79  escape_stmt                       | 199
 80  expression                         | 42 57 60 73 74 81 84 85 107 128 129
133 136 148 175 189 192 222 228
 81  expression_or_wild                | 134 225
 82  extent_reference                  | 153 197
```

```
 83  factor                          | 214
 84  foreach_expr                    | 88
 85  forloop_expr                    | 88
 86  formal_parameter                | 92 160
 87  forward_path_qualifier          | 154
 88  for_expr                        | 133 192
 89  from_clause                     | 206
 90  function_call                   | 162
 91  function_decl                   | 65 180 185
 92  function_head                   | 91
 93  function_id                     | 92 169
 94  generalized_types               | 150
 95  general_aggregation_types       | 94
 96  general_array_type              | 95
 97  general_attribute_qualifier     | 163
 98  general_bag_type                | 95
 99  general_list_type               | 95
 100 general_or_map_call             | 162
 101 general_ref                     | 42 100
 102 general_schema_alias_id         | 184
 103 general_schema_ref              | 184
 104 general_set_type                | 95
 105 generic_type                    | 94
 106 group_qualifier                 | 132 163
 107 identified_by_clause            | 67 206
 108 if_expr                         | 133
 109 if_stmt                         | 199
 110 increment                       | 111
 111 increment_control               | 170
 112 index                           | 113 114
 113 index_1                         | 115
 114 index_2                         | 115
 115 index_qualifier                 | 132 163
 116 instantiation_foreach_control   | 117
 117 instantiation_loop_control      | 76
 118 integer_type                    | 193
 119 interval                        | 192
 120 interval_high                   | 119
 121 interval_item                   | 119
 122 interval_low                    | 119
 123 interval_op                     | 119
 124 label                           | 72
 125 list_type                       | 39
 126 literal                         | 157
 127 local_decl                      | 40
 128 local_variable                  | 127
 129 logical_expression              | 64 72 108 109 153 164 220 232
 130 logical_literal                 | 126
 131 logical_type                    | 193
 132 map_attribute_declaration       | 64 138
 133 map_attr_assgnmt_expr           | 55 56 108 132
 134 map_call                        | 100
```

```
135 map_decl                          | 180
136 map_decl_body                     | 135
137 map_id                            | 66 135 139
138 map_project_clause                | 69 76 136
139 map_ref                           | 134 140 174
140 map_reference                     | 141
141 map_subtype_of_clause             | 66 135
142 multiplication_like_op            | 214
143 named_types                       | 45 150
144 null_stmt                         | 199
145 number_type                       | 193
146 numeric_expression                | 48 49 110 112 156 172 233
147 one_of                            | 208
148 parameter                         | 34
149 parameter_id                      | 86 198 212
150 parameter_type                    | 38 86 92 96 98 99 104 128
151 partition_id                      | 70 203 206
152 partition_qualification           | 134 225
153 path_condition                    | 43 87
154 path_qualifier                    | 163
155 population                        | 162
156 precision_spec                    | 165
157 primary                           | 192
158 procedure_call_stmt               | 199
159 procedure_decl                    | 65 180 185
160 procedure_head                    | 159
161 procedure_id                      | 160 169
162 qualifiable_factor               | 157
163 qualifier                         | 42 157
164 query_expression                  | 192
165 real_type                         | 193
166 reference_clause                  | 182 187
167 rel_op                            | 168
168 rel_op_extended                   | 80
169 rename_id                         | 173
170 repeat_control                    | 85 117 171
171 repeat_stmt                       | 199
172 repetition                        | 73
173 resource_or_rename                | 166
174 resource_ref                      | 173
175 return_stmt                       | 199
176 rule_decl                         | 180 185
177 rule_head                         | 176
178 rule_id                           | 177
179 schema_id                         | 102
180 schema_map_body_element           | 181
181 schema_map_body_element_list      | 182
182 schema_map_decl                   | 209
183 schema_map_id                     | 102 182
184 schema_ref_or_rename              | 166
185 schema_view_body_element          | 186
186 schema_view_body_element_list     | 187
```

```
187 schema_view_decl                  | 209
188 schema_view_id                    | 102 187
189 selector                          | 55 58
190 set_type                          | 39
191 simple_expression                 | 37 80 120 121 122 146
192 simple_factor                     | 83
193 simple_types                      | 45 71 150
194 skip_stmt                         | 199
195 source_attribute_reference        | 116
196 source_entity_reference           | 82
197 source_parameter                  | 89
198 source_parameter_id               | 71 197
199 stmt                              | 54 58 59 91 109 159 171 176
200 string_literal                    | 126
201 string_type                       | 193
202 subtype_constraint                | 207
203 subtype_partition_header          | 135 226
204 supertype_expression              | 147 202 208
205 supertype_factor                  | 204
206 supertype_partition_header        | 135 226
207 supertype_rule                    | 226
208 supertype_term                    | 205
209 syntax                            |
210 target_entity_reference           | 64 211
211 target_parameter                  | 66 135
212 target_parameter_id               | 211 213
213 target_parameter_ref              | 132
214 term                              | 191
215 type_id                           | 169
216 type_label                        | 38 105
217 type_label_id                     | 216
218 type_reference                    | 71
219 unary_op                          | 192
220 until_control                     | 170
221 variable_id                       | 84 111 116 128 164
222 view_attribute_decl               | 224
223 view_attribute_id                 | 222
224 view_attr_decl_stmt_list          | 228
225 view_call                         | 162
226 view_decl                         | 180 185
227 view_id                           | 226
228 view_project_clause               | 226
229 view_reference                    | 82 225 230
230 view_subtype_of_clause            | 226
231 where_clause                      | 67 84 176 203 206
232 while_control                     | 170
233 width                             | 234
234 width_spec                        | 46 201
```

Annex C
(normative)
EXPRESS-X to EXPRESS Tranformation Algorithm

This annex describes how a collection of view declarations may be transformed into a collection of EXPRESS entity declarations suitable for representing the results of an EXPRESS-X execution. The transformation is described as an algorithm taking the text of a view declaration as input and producing the text of an entity declaration as output. The algorithm is given here for specification purposes only, not to prescribe a particular implementation.

The transformed entities are assumed to exist in a uniquely named schema, into which all necessary foreign declarations have been interfaced.

Algorithm:

a) If the view declaration is a SELECT view (i.e., does not define any view attributes), skip the declaration.

b) Change the keyword VIEW to ENTITY.

c) Delete entirely any FROM ,WHERE, and/or IDENTIFIED_BY clauses. Delete only WHERE clauses in the header; do not delete constraint where clauses.

d) Delete the keyword SELECT.

e) If the view declaration contains partitions, delete entirely all but the first partition declaration, and delete the keyword PARTITION and the partition identifier (if any) from the first partition declaration.

f) Delete the assignment operator and expression for each view attribute.

g) Change the keyword END_VIEW to END_ENTITY.

EXAMPLE 4 —

```
VIEW a ABSTRACT SUPERTYPE;
PARTITION one:
FROM b:one, c:two
WHERE cond1;
      cond2;
SELECT
  x : attr1 := expression1;
  y : attr2 := expression2;
PARTITION two:
FROM d:two, e:three
WHERE cond3;
      cond4;
SELECT
  x : attr1 := expression3;
  y : attr2 := expression4;
END_VIEW;
```

is transformed into the following EXPRESS entity declaration:

```
ENTITY a ABSTRACT SUPERTYPE;
  x : attr1;
  y : attr2;
END_ENTITY;
```

EXAMPLE 5 —

```
VIEW b SUBTYPE OF (a);
PARTITION one:
WHERE cond5;
SELECT
  z : attr3 := expression5;
PARTITION two:
WHERE cond6;
SELECT
  z : attr3 := expression6;
WHERE
  WR2 : rule_expression2;
END_VIEW;
```

is transformed into the following EXPRESS entity declaration:

```
ENTITY b SUBTYPE OF (a);
  z : attr3;
WHERE
  WR2 : rules_expression2;
END_ENTITY;
```

Annex D
(informative)
Implementation considerations

D.1 Push mapping

An implementation shall be said to be a push mapping implementation if it meets all of the following criteria:

– The mapping engine accepts one or more source data sets, and produces one or more output data sets.

– The output data sets are derived from the input data sets by the execution and evaluation of all of the VIEW and MAP declarations.

– Every instance in the source data sets is mapped as specified in the mapping schema into the output data sets.

D.2 Pull mapping

An implementation shall be said to be a pull mapping implementation if it meets all of the following criteria:

– The mapping engine accepts one or more source data sets.

– Specified target data instances, and only those specified, are derived on demand from the input data sets by the execution and evaluation of the appropriate VIEW or MAP declarations.

NOTE — This part of ISO 10303 does not define how VIEW / MAP declarations are selected for pull mapping.

D.3 Support of constraint checking

An implementation shall be said to support constraint checking if it implements the concepts described in clause 9.6 of ISO 10303-11:1994 against entity instances in target populations and against view instances in the view extents.

NOTE — The evaluation of constraints has no effect on execution.

Propagation of updates is not possible in situations where any of the following hold:

– The view / target entity is derived from / mapped to two or more source entities by applying a join operation. (For example, the view / target entity `person_in_dept` corresponds to the source entities `person` and `department` where the join condition `person.id = depart-`

`ment.person_id` evaluates to true.)

– Duplicates (with respect to value equivalence of attributes) which exist in the source data are eliminated in the view / target data.

– View / target attributes are derived from / mapped to source schema elements by applying mathematical expressions that are not mathematically invertible.

– The view / target schema defines additional subtypes which do not exist in the source schema(s).

– Subtypes which are defined in the source schema(s) are projected (i.e., not contained) in the view / target schema.

– The sort order of source attributes of type AGGREGATE is eliminated in the view / target schema.

– Duplicates (with respect to value equivalence) of elements of source attributes of type AGGREGATE are eliminated in the view / target schema.

– A single source entity corresponds to a network of interconnected view / target entities (by relationships or equivalence of attribute values[1]).

1. The latter kind of relationship is comparable to primary key - foreign key relationships in the relational data model.

Annex E
(informative)
Path operator reference functions

The following implements unnest, an EXPRESS function referred to in clause 10.8 and clause 10.9.

```
FUNCTION unnest(src : GENERIC) : AGGREGATE OF GENERIC;
LOCAL
  result : AGGREGATE OF GENERIC := [];
  tmp : AGGREGATE OF GENERIC;
END_LOCAL;
  IF ['LIST', 'BAG', 'SET', 'ARRAY', 'AGGREGATE'] IN
     TYPEOF(src)
  THEN
    REPEAT i := 1 TO HIINDEX(src)
      IF ['LIST', 'BAG', 'SET', 'ARRAY', 'AGGREGATE']
    * TYPEOF(src[i]) > 0
      THEN  -- aggregate type element
  tmp := unnest(src[i]);
  REPEAT j := 1 TO HIINDEX(tmp);
    result := result + tmp[i];
  END_REPEAT;
      ELSE
  IF SIZEOF(['STRING', 'BINARY', 'BOOLEAN', 'NUMBER', 'BOOLEAN']
      * TYPEOF(src[i])) = 0
  THEN  -- entity instance element
    result := result + src[i];
  END_IF;
      END_IF;
    END_REPEAT;
  ELSE
    IF SIZEOF(['STRING', 'BINARY', 'BOOLEAN', 'NUMBER', 'BOOLEAN']
              * TYPEOF(src)) = 0
    THEN  -- entity instance
      result := [src];
    END_IF;
  END_IF;
  RETURN (result);
END_FUNCTION;
```

Bibliography

EXPRESS-V language (ISO TC184/SC4/WG5 N251).

EXPRESS-M language (ISO TC184/SC4/WG5 N243).

BRITTY language.

Wirth, Niklaus, *"What can we do about the unnecessary diversity of notations for syntactic definitions?,"* Communications of the ACM, November 1977, v. 20, no. 11, p. 822.

Index